SKI: Exposing Kernel Concurrency Bugs through Systematic Schedule Exploration

Pedro Fonseca, *Max Planck Institute for Software Systems (MPI-SWS);*
Rodrigo Rodrigues, *CITI/NOVA University of Lisbon;*
Björn B. Brandenburg, *Max Planck Institute for Software Systems (MPI-SWS)*

https://www.usenix.org/conference/osdi14/technical-sessions/presentation/fonseca

This paper is included in the Proceedings of the
11th USENIX Symposium on
Operating Systems Design and Implementation.

October 6–8, 2014 • Broomfield, CO

978-1-931971-16-4

SKI: Exposing Kernel Concurrency Bugs through Systematic Schedule Exploration

Pedro Fonseca
Max Planck Institute for Software Systems (MPI-SWS)

Rodrigo Rodrigues
NOVA University of Lisbon (CITI/NOVA-LINCS)

Björn B. Brandenburg
Max Planck Institute for Software Systems (MPI-SWS)

Abstract

Kernel concurrency bugs are notoriously difficult to find during testing since they are only triggered under certain instruction interleavings. Unfortunately, no tools for systematically subjecting kernel code to concurrency tests have been proposed to date. This gap in tool support may be explained by the challenge of controlling precisely which kernel interleavings are executed without modifying the kernel under test itself. Furthermore, to be practical, prohibitive runtime overheads must be avoided and tools must remain portable as the kernel evolves.

In this paper, we propose SKI, the first tool for the systematic exploration of possible interleavings of kernel code. SKI finds kernel bugs in *unmodified* kernels, and is thus directly applicable to different kernels. To achieve control over kernel interleavings in a portable way, SKI uses an adapted virtual machine monitor that performs an efficient analysis of the kernel execution on a virtual multiprocessor platform. This enables SKI to determine which kernel execution flows are eligible to run, and also to selectively control which flows may proceed. In addition, we detail several essential optimizations that enable SKI to scale to real-world concurrency bugs.

We reliably reproduced previously reported bugs by applying SKI to different versions of the Linux kernel and to the FreeBSD kernel. Our evaluation further shows that SKI was able to discover, in widely used and already heavily tested file systems (e.g., ext4, btrfs), several unknown bugs, some of which pose the risk of data loss.

1 Introduction

In the current multi-core era, kernel developers are under permanent pressure to continually increase the performance of kernels through concurrency. Examples of such efforts include reducing the granularity of locking [59], rewriting subsystems to use parallel algorithms [26], and using non-traditional and optimistic synchronization primitives (such as RCU [52] and lock-free data structures [67]). Unfortunately, previous experience has shown that all these efforts are error-prone and can easily lead to kernel concurrency bugs — bugs that are only exposed by a subset of the possible thread interleavings.

In practice, kernel developers find concurrency bugs mostly through manual code inspection [39, 69] and stress testing [14, 64] (i.e., applying intense workloads to increase the chances of triggering concurrency bugs). While useful, both approaches have significant shortcomings: code inspection is labor-intensive and requires significant skill and experience, and stress testing, despite having low overhead and being amenable to automation, offers no guarantees and can easily fail to uncover difficult to find concurrency bugs — i.e., edge cases that are only triggered by a tiny subset of the interleavings. It thus stands to reason that kernel developers could benefit from tools without these limitations.

To this end, we propose a *complementary* testing approach for automatically finding kernel concurrency bugs. Our approach explores the kernel interleaving space in a systematic way by taking full control over the kernel thread interleavings. Similar approaches have been explored for user-mode applications, yielding good results [20, 53, 54], but have not yet been applied to commodity kernels because achieving control over the thread interleavings of kernels involves several challenges. First, to be practical, a concurrency testing tool must be generally applicable, rather than being specific to a particular kernel or kernel version, which precludes kernel-specific modifications. Second, the kernel is the software layer that implements its own thread scheduler, as well as the thread abstraction itself, making the external control of thread interleavings non-trivial. Finally, to be effective, such a tool must be able to control kernel interleavings while introducing a low overhead.

In this paper, we report on the design and an evaluation of SKI[1], the first tool for the systematic exploration of kernel interleavings to overcome these challenges. To achieve control over kernel interleavings in a portable way, SKI uses an adapted virtual machine monitor that (1) determines the status of the various threads of execution, in terms of being blocked or ready to run, to understand the scheduling restrictions, and (2) selectively blocks a subset of these threads in order to enforce the desired schedule. Notably, these key tasks are achieved without any modification to the kernel and without specific knowledge of the semantics of the kernel's internal synchronization primitives. Furthermore, we propose several optimizations, both at the algorithmic and at the implementation levels, that we found to be important for scaling SKI to real-world concurrency bugs.

We evaluated SKI by testing several file systems in recent versions of the Linux kernel and we found 11 previously unknown concurrency bugs. Of these, several concurrency bugs can cause serious data loss in important file systems (*ext4* and *btrfs*). We also show how SKI can be used to reproduce concurrency bugs that have been previously reported in two different operating systems (Linux and FreeBSD), and compare SKI's performance against the traditional stress testing approach.

We believe that SKI is an important step towards increased kernel reliability on multicore platforms. Nonetheless, there remains significant room for exploiting domain- and kernel-specific knowledge. For instance, in this paper we propose a scheduling algorithm (for performing the schedule exploration) that is generic in the sense that it makes no assumptions about the kernel under test. However, based on the SKI infrastructure, other kernel-specific scheduling algorithms could be implemented, for example, to restrict the interleavings explored to those that affect specific kernel instructions, such as code that was recently modified. Thus, we believe that SKI can provide benefits even beyond those described in this paper, since it can serve as an experimentation framework for different systematic techniques.

The rest of the paper is organized as follows. Section 2 motivates the need for better kernel testing tools. Section 3 presents the design of SKI. Section 4 proposes several optimizations to make SKI scale to real-world concurrency bugs. Section 5 describes the details of our implementation. Section 6 evaluates SKI and Section 7 discusses some of its limitations. In Section 8 we discuss related work and finally we conclude in Section 9.

2 Systematic testing

A *systematic* exploration of the interleaving space, in contrast with a stress testing approach, relies on judiciously controlling the thread schedule for each execution of the software under test to maximize the coverage of the interleaving space.

At a high level, systematic approaches increase the effectiveness of testing by avoiding redundant interleavings and prioritizing interleavings that are more likely to expose bugs, e.g., those that differ more from interleavings that have already been explored. It has been shown both analytically and empirically that such methods offer better results than traditional *ad hoc* approaches [20].

To achieve this level of control over interleavings, systematic approaches rely on a custom thread scheduler that implements two basic mechanisms. The first mechanism infers thread liveness to understand which schedules it can choose, which can be achieved by intercepting and understanding the semantics of the synchronization functions. The second mechanism overrides the regular scheduler by allowing only a specifically chosen thread to make progress at any point in time.

In the case of user-mode applications, both of these two essential mechanisms can be easily implemented in a proxy layer (e.g., through *LD_PRELOAD* or *ptrace*) by intercepting all relevant synchronization primitives to infer and override the liveness state of each thread [20, 53, 54]. Unfortunately, a direct application of the user-mode method to the kernel would require modifying the kernel itself, which would suffer from several disadvantages:

- **Lack of portability and API instability.** Any dependency on kernel-internal APIs would a priori limit the portability of the envisioned testing tool, preventing its seamless application across different kernels and even across different versions of the same kernel. In contrast to well-documented, standardized user-space interfaces (e.g., the pthreads API), the internal API of most kernels is not guaranteed to be stable, and in fact typically changes from version to version. In particular, given the current trend towards increased hardware parallelism, kernel synchronization has generally been an active area of development in Linux and other kernels [26, 52].

- **Complexity of the internal interface.** An additional problem with the internal API of the kernel, also noted in previous work [32], is that the semantics of in-kernel synchronization operations are particularly complex. Furthermore, the exact semantics of such operations tend to differ from kernel to

[1] Systematic Kernel Interleaving explorer

kernel. This calls for solutions that do not require a detailed understanding of these semantics.

- **Other forms of concurrency.** Interrupts are pervasive and critical to kernel code. However, exercising fine-level control over their timing from within the kernel itself would be particularly challenging, as interrupts are scheduled by the hardware.[2]
- **Intrusive testing.** Requiring modifications to the tested software goes against the principles of testing [43] — testing *modified* versions of the software can potentially introduce or elide bugs.

In the next section we explain how SKI overcomes these challenges while enabling the systematic exploration of kernel thread interleavings.

3 SKI: Exploring kernel interleavings

This section presents the design of SKI. We start by providing an overview of our solution (Section 3.1), and then we describe how SKI exercises control over thread interleavings (Section 3.2) and how it gathers the necessary liveness information (Section 3.3). We conclude this section with a description of the scheduling algorithms employed, i.e., the interleavings chosen for each run (Section 3.4).

3.1 Overview

The inputs given to SKI are the initial state of the system under test and the kernel input that is to be tested concurrently (i.e., two or more concurrent system calls). Given these inputs, SKI carries out several test runs corresponding to different concurrent executions, where each test run is fully serialized, i.e., the tool enables only a single thread to execute at each instant. This enables precise control over which interleavings are executed, and allows SKI's scheduler to choose successive runs to improve the interleaving space coverage. Either during or after each test run, a bug detector is used to determine if the test has flagged a possible bug. Such bug detectors can perform simple, generic actions like detecting crashes, or complex, application-specific actions like running a system integrity check after the test run.

As mentioned in the previous section, for SKI's scheduler to gain control over the interleavings executed by the kernel, it must perform two key tasks: inferring thread liveness and overriding the scheduler. To accomplish both without modifying the OS kernel under test, we implement the scheduler of SKI at the level of a modified virtual machine monitor (VMM), taking as input a virtual machine (VM) image that incorporates the initial state of the kernel immediately before the system calls are invoked concurrently. Implementing the scheduler at the VMM level enables it to both observe and control the kernel under test.

This advantage comes, however, at the cost of making it more difficult to implement the two aforementioned key tasks. This is because, at the VMM level, the hypervisor observes a stream of machine instructions to be executed, and has direct access only to the physical resources of the underlying hardware (such as registers or memory contents). These low-level concepts are distant from the abstractions that are implemented by the kernel in software, such as threads and their respective contexts. Furthermore, it would intuitively seem necessary to have access to these abstractions for suspending the execution of a thread and replacing it with another thread.

3.2 Exercising control over threads

To control the progress of threads, SKI relies on the observation that the most widely used kernels (e.g., Linux, Windows, MacOS X, FreeBSD) include a mechanism to allow applications to *pin* threads to individual CPUs (i.e., to specify the thread affinity). This mechanism, provided by kernels to user-mode applications for performance reasons, can be exploited to create a 1:1 mapping between threads (a kernel abstraction) and virtual CPUs (an ISA component, controllable by the VMM). This mapping in turn allows SKI to block and resume a thread execution by simply suspending and resuming the corresponding virtual CPU's execution of machine instructions.

Apart from the user-space threads that invoke system calls, operating systems have another type of threads, which similarly execute kernel code, namely *kernel threads* [7]. Kernel threads are used by the kernel to asynchronously execute tasks. Despite not being associated with user-mode processes, some kernel threads can be pinned to different CPUs from user-space. For kernel threads that cannot be pinned to other CPUs for OS-specific reasons, SKI is not able to explicitly control their schedule and therefore lets the OS schedule them.

To implement the mapping between threads and CPUs, SKI includes, in addition to the modified VMM, a user-mode component that runs inside the VM and issues system calls to pin threads to virtual CPUs (see Sec-

[2] While user-mode signals are similar to interrupts, many programs do not use them and therefore existing user-mode tools do not handle them [20, 53].

tion 5.3). Note that for scalability reasons, each test generally involves only few threads, and hence it suffices to configure a small number of virtual CPUs.

3.3 Inferring liveness

To explore the interleaving space, SKI requires information about whether threads are blocked or able to progress, analogously to what is required by the existing user-mode tools [20, 53, 54]. This requires SKI to be able to identify constructs such as spin-locks or barriers, where a CPU executes a tight loop, constantly checking the value of a memory location for changes. SKI would be impractical if it were not able to detect such constructs, for several reasons. First, executions would take longer because more instructions would be executed (e.g., iterations of a spin loop). Second, because more instructions would be executed, the space of possible interleavings would significantly increase, since the number of possible interleavings is exponential in the length of the test. Third, and most importantly, given the scheduling algorithm that we describe in Section 3.4, two interleavings could be considered different even when they only differ in the number of iterations executed by the polling loop of a spin lock. This would be detrimental to the efficiency of SKI, since many of the explored schedules would be effectively equivalent.

The difficulty in inferring thread liveness is that, from the point of view of the VMM, CPUs are constantly executing instructions. As such, it is difficult to distinguish the normal execution of a program from a polling loop.

One possible solution that we considered, but ultimately rejected, relies on annotating the kernel by specifying the locations within the kernel code where the CPU executes instructions without making any actual progress, namely situations where the kernel is waiting for some event external to the CPU (such as an action performed by some other CPU or a device notification). However, this approach would be laborious, error prone, and non-portable.

Instead, we found several simple heuristics independent of the kernel code that enable the VMM to infer whether a CPU is making progress or not.

H1: Halt heuristic. The first heuristic flags the CPU as non-live when it executes the halt instruction (HLT).[3] According to the instruction set specification, HLT marks the CPU as waiting for interrupts. This instruction is typically used by kernels to implement, in an energy efficient way, the idle thread when the kernel scheduler has no other threads to run. When the CPU subsequently receives an interrupt, it is marked as live again.

H2: Pause heuristic. The second heuristic relies on the observation that kernels use the pause instruction (PAUSE) to efficiently implement spin-locks. In the x86 architecture, the pause instruction has been introduced to avoid wasting bandwidth on the memory bus when a CPU goes into a tight polling loop, and therefore its execution is a good indication that the CPU is spinning on a lock. Thus, when our modified VMM detects the execution of two nearby pause instructions, i.e., within an instruction window of size h_2, it considers the CPU to be non-live and takes note of the memory read-set associated with the instructions executed between the two pause instructions. Pause instructions in close proximity are detected by the VMM by checking, at every pause instruction, whether another pause instruction was recently executed. Later on, when another CPU changes one of the addresses in the read-set, the non-live CPU is optimistically marked as live again.

H3: Loop heuristic. The third heuristic detects situations where the CPU is waiting for some external event, but that are not caught by the second heuristic. This could happen if, for example, a spin-lock were implemented without including the pause instruction. To detect CPUs stuck in a polling loop, our modified VMM maintains a window, of size h_3, of the last few instructions executed by each CPU. If a CPU repeatedly executes the same instructions (i.e., if it executes a loop), and if an instruction in the loop repeatedly reads the same value from the same memory address, the executing context is flagged as non-live after a certain number of loop iterations. Again, SKI takes note of the read-set of detected polling loops to later re-enable the CPU.

H4: Starvation heuristic. As a last resort, in case the above heuristics are not able to detect situations where there is no progress, SKI keeps a count of the number of instructions executed continuously by the current CPU, and, if it exceeds a threshold (h_4), it conservatively presumes that the CPU is no longer making progress. The CPU is marked live again after a certain number of instructions have been executed by the other CPUs. This heuristic ensures the detection, for example, of loops that are missed by H3 if h_3 is set smaller than the loop size.

We determined the values for the thresholds of these heuristics, which remained constant throughout all our tests, through simple experimentation. From our experience, these mechanisms were sufficient to ensure the effectiveness of SKI for a wide range of kernel versions, at both reproducing previously known bugs and at finding unknown bugs.

[3] We focus on the ubiquitous x86 architecture in this paper; the presented ideas, however, can be similarly applied to other architectures.

3.4 Scheduling algorithm

SKI executes a VM multiple times under different schedules to ensure interleaving diversity across the runs. To select and prioritize the interleavings that are to be explored SKI needs to implement a scheduling algorithm.

SKI uses an extension of the PCT algorithm [20], a state-of-the-art algorithm originally developed for user-mode applications, which has been shown to be effective at uncovering user-mode concurrency bugs. SKI extends PCT by supporting interrupts (Section 3.4.2), which is a fundamental requirement for testing operating systems.

Nonetheless, we consider the proposed algorithm to be just one instance from a range of possible algorithms (albeit one that in our experience happens to work well), and developers that make use of the tool might consider adding other, more refined algorithms. For example, it may be possible to develop effective scheduling algorithms that exploit specific characteristics of kernel code.

Since the SKI scheduler must handle both threads and interrupts, it schedules *contexts* instead of threads; we will thus refer to contexts throughout this description.

3.4.1 Background: PCT algorithm

Conceptually the scheduler executes instructions sequentially one by one; that is, at any point during the execution, only one of the live contexts is allowed to progress, and the eligibility of the context to execute another instruction is re-evaluated after each instruction. Through this process, the scheduler is able to effectively control the chosen interleaving. In practice, however, our implementation optimizes this process by using a JIT compiler and by only introducing checks as needed (Section 5).

A strawman design for the scheduler would be to use a fixed ordering of the various contexts, and to run the first context for the longest possible period until it is no longer able to run. At this point, the scheduler chooses the second context to run until either it is also no longer able to run, or the first context becomes able to run again, and so on. While this initial design suffices to create valid schedules and allows tests to finish, it does not create a diverse set of schedules.

To achieve a good diversity of schedules across different runs, the scheduler uses two strategies. The first is to randomly assign initial priorities to the contexts, and use these priorities instead of a fixed order to determine the context that should run at each instant – this is the context with the highest priority among those that are not blocked. The second strategy consists of reducing, at random points during the execution of a test, the priority of the context that is scheduled. If the priority decrease is large enough, this will cause another context to become the one with the highest priority, and therefore this other context will be scheduled to run. By varying both the initial priorities and the location of such *reschedule points* in a controlled way, the scheduler is able to control the range of tested schedules.

The reschedule points are chosen prior to each run by randomly selecting a set of offsets from the start of the test (in terms of the total number of instructions executed) within a certain range. Then, during the execution, whenever the total number of instructions executed reaches one of these offsets, the priority of the currently scheduled context is lowered so that it becomes the lowest-priority context, and thus another runnable context is selected for execution in the next step.

The set of reschedule points is determined according to two parameters: the expected number of execution steps k and the desired number of reschedule points p, with the simple interpretation that there will be up to p reschedules within the first k instructions of the execution of the test (and none thereafter, should the test execute for more than k instructions). That is, for a given k and p, the set of p reschedule points is selected by choosing uniformly at random p offsets from the range $[0,k]$.

3.4.2 Handling interrupts

Given that SKI operates at the level of the virtual machine monitor, it does not have access to the thread abstraction that is used by schedulers for user-mode applications. Thus, instead of scheduling threads, our algorithm schedules CPUs. In addition, another distinction to user-mode schedulers is that the scheduler needs to make decisions regarding when interrupts should be dispatched. Interrupts do not appear in the context of user-mode programs, but we need to control their schedule when testing the kernel for two different reasons. First, concurrency bugs may depend on the interleaving of interrupts, so our algorithm should be able to explore this part of the interleaving space. Second, interrupts are in some cases required for the successful completion of system calls, and therefore interrupts need to be scheduled to conclude the execution of the tests. For example, some system calls are only able to finish if, during their execution, other CPUs handle the TLB flush interrupt.

As the scheduler needs to consider when interrupts are handled, each CPU is tracked as being in one of two different contexts: it may either execute in the context of an interrupt handler (*interrupt-context*), or it may execute outside of the context of any interrupt handlers (*CPU-context*). Each *interrupt-context* is defined by the CPU on which it arrived and by the interrupt number

Schedule 1			Schedule 2		
CPU1	INT1	CPU2	CPU1	INT1	CPU2
INST 1			INST 1		
INST 2			INST 2		
INST 3			<**inv**>		
INST 4				INST 1	
<**end**>				INST 2	
	INST 1			<**end**>	
	INST 2				INST 1
	<**end**>				INST 2
		INST 1			INST 3
		INST 2			<**end**>
		INST 3	INST 3		
		<**end**>	INST 4		
			<**end**>		

Figure 1: Two examples illustrating schedules produced by SKI. Each schedule involves three contexts, two CPU-contexts and one interrupt-context. Both schedules start with the same initial context priorities. However, Schedule 2 differs from Schedule 1 because it contains one priority inversion (<inv>).

that it represents. From the point of view of the scheduler, *interrupt-contexts* are created, and therefore become schedulable, when the corresponding interrupt arrives on its specific CPU. These execution contexts are, to our scheduler, the equivalent to threads for other systematic exploration algorithms, and as such they need to be detected by the scheduling logic. SKI infers the context by tracking the interrupt handler dispatches and the *IRET* instruction invocations (which are used to return from interrupt handlers). Figure 1 shows examples of schedules involving two *CPU-contexts* and one *interrupt-context*.

To achieve further control over the tests, SKI allows the user to specify a set of execution contexts that are allowed to run during the test. In particular, placing restrictions on the set of eligible execution contexts may be useful in specific testing scenarios, to restrict the scheduling space that is explored.

3.5 Discussion

The design of SKI ensures correctness, meaning that SKI never causes the kernel to exhibit a behavior that could not possibly occur during normal executions of the kernel, because SKI exercises control over the kernel schedule by temporarily suspending the execution of instructions on chosen CPUs. Correct kernels have to be able to handle this mechanism because the hardware specification does not provide guarantees about the speed of the CPUs. Furthermore, modern kernels are expected to work well within virtual machines, where the apparent speed of CPUs is not guaranteed to be regular simply because the host system might be under heavy load.

Despite this correctness guarantee, some bug detectors may still produce false positives (e.g., data race detectors). In such cases and regardless of how the interleaving space is explored, the obtained results require further analysis specific to the the employed bug detector.

4 Efficiency: Scaling to real code

The total number of possible schedules grows exponentially with the length of the code under test. For most programs, including the kernel, it is not practical to *exhaustively* explore all interleavings, and therefore it is important for concurrency testing tools to include mechanisms for increased scalability.

The p parameter, used by the scheduling algorithm (Section 3.4), constrains the schedules that may be explored and therefore improves scalability by bounding the number of possible schedules. This is done without much impact on the effectiveness of the testing tool, given the observation that, in practice, most bugs can be triggered with few reschedule points [20]. Similarly, it has been shown that many concurrency bugs can be triggered with a small number of threads [48] and with a small number of concurrent requests [60]. Based on these observations, we configured SKI in our tests to use small values for these three dimensions (reschedule points, number of CPUs, and number of system calls).

Despite these optimizations, we noticed in our initial tests that SKI's scalability was limited by the fact that even a single system call can execute a large number of instructions — typical system calls execute many thousands or even millions of instructions. This implied that, even if we limited SKI to $p = 1$, the number of runs that would be required to explore all schedules were on the same order of magnitude as the number of instructions.

To address this scalability issue, SKI relies on a technique first proposed by Godefroid [36] that exploits the fact that some schedules are equivalent and thus redundant, as illustrated in Figure 2. In particular, we rely on the observations that (1) schedules that do not differ in terms of the relative order of communication points (where threads see the effects of each other) are observationally equivalent from the standpoint of the interleaved threads, and that (2) most of the kernel instructions do not constitute communication points between

Schedule 1		Schedule 2		Schedule 3	
CPU1	CPU2	CPU1	CPU2	CPU1	CPU2
A=1		A=1		A=1	
<**inv**>		B=1		B=1	
	A=0	<**inv**>		C=B+A	
	D=A+1		A=0	<**inv**>	
B=1			D=A+1		A=0
C=B+A		C=B+A			D=A+1
PRINT C		PRINT C		PRINT C	

Figure 2: Example showing two equivalent schedules (Schedules 1 and 2) and one schedule that is not equivalent to either of the others (Schedule 3). In this example, only variable A is used for communication between CPUs. Because variable B is accessed by only one CPU, placing the priority inversion point (<inv>) immediately before (Schedule 1) or immediately after (Schedule 2) the statement B=1 does not change the result of the execution.

CPUs. Taken together, these two observations allow us to significantly improve SKI's scalability by restricting reschedules to occur only at communication points.

More precisely, we define a *point of communication* as an instruction that accesses a memory location that is also accessed by another CPU during the test, and where at least one of the accesses is a write. Such concurrent memory accesses can influence the final outcome of the execution: in the case of two concurrent writes, the last value to be written prevails, and in the case of a write concurrent with a read, the value read may or may not reflect the write, depending on the schedule. Prior tools have also tried to avoid equivalent schedules, but rely instead on identifying and preempting threads at either possible data races or the invocation of synchronization primitives [53].

SKI gathers the location of possible communication points by monitoring memory accesses during the tests. During each run, it tracks the locations of the memory accesses, the CPU responsible for the accesses, and the types of accesses (read or write). After each run, SKI generates a set of program addresses that are potential communication points, and merges this information with an accumulated set of potential communication points for that specific test case. Note that this process does not rely on sample runs — every run monitors the memory accesses and, therefore, potentially learns new communication points. As this accumulated set is constructed, it is used in subsequent runs for the same test case to decide which schedules are equivalent, thereby limiting the set of instructions that qualify as reschedule points.

In our experiments, we observed that, as expected, both data and synchronization accesses were identified as communication points. To give some examples, data accesses occur when both CPUs try to modify the same field in a shared structure (e.g., a file reference count), and synchronization accesses occur when both CPUs try to acquire the same lock. An advantage of SKI's dynamic approach is that whether or not an instruction qualifies as a reschedule point depends on the code that *both* CPUs actually execute (e.g., the specific system calls or interrupt handlers that are invoked). As a result, if two CPUs acquire different locks unrelated to the tested functionality, such accesses will not be considered communication points (in the context of the current test case).

In practice, SKI estimates the expected number of instructions, k (recall Section 3.4), based on previous runs. With the communication points optimization, instead of considering individual instructions when placing reschedule points, we consider only communicating instructions, and thus let the algorithm take coarser-grained steps in its exploration of the interleaving space. That is, by limiting the set of reschedule point candidates, the magnitude of the parameter k is effectively reduced. In addition to these algorithmic optimizations, SKI includes several optimizations, at the level of the implementation, to ensure its effectiveness (Section 5.4).

5 Implementation

We implemented SKI by modifying QEMU, a mature and open-source VMM, and its JIT compiler. In total, our implementation added $13,542$ lines of source code to QEMU. We also built a user-mode testing framework consisting of 674 lines of source code to help users write test cases for SKI (Section 5.3). In addition, we implemented various scripts to set up and automate tests and also to analyze the gathered information.

5.1 Overview

SKI provides a helper tool to allow kernel developers to specify the concurrent system calls, by building a VM containing the corresponding test case (Section 5.3). When executed under SKI, this VM first goes through an initialization phase, performing test-specific actions to configure the system, and then signals the beginning of the test to the VMM using hypercalls (i.e., calls between the VM and the VMM). When all virtual CPUs have received the signal, the SKI scheduler is activated.

SKI's first action is to take a snapshot of the VM. The VM snapshot includes the entire machine state (memory state, disk state, CPU state, etc.) and thus allows SKI to run multiple executions from an identical initial state.

Starting from this VM snapshot, SKI places reschedule points and assigns starting priorities as described in Sections 3.4 and 4, and then resumes the execution of the highest-priority context and enforces the chosen schedule, thereby exploring different schedules on each run.

To mark the end of the test, the user-mode component inside the VM issues a hypercall to the VMM. Afterwards, the VM is allowed to run normally (i.e., without schedule restrictions) until the testing application asks to terminate the execution. This last phase is useful to let the user-mode component execute test-specific diagnostics (such as a file system check) inside the VM.

5.2 Runnable contexts

The scheduler of SKI allows, at any point in time, only the *live* and *active* context with the highest priority to run. The liveness of a context is inferred by the VMM according to the heuristics explained in Section 3.3; the criteria for determining whether a context is active or not depends on the type of context. A CPU-context is considered active if it has not reached the end of the test, which is flagged by the user-mode component using a hypercall, as discussed above, whereas an interrupt-context is considered active only after it has been triggered by the respective hardware device and before the corresponding IRET instruction has been executed.

5.3 Helper testing framework

We built a user-mode helper framework that allows users to easily build a testing VM ready to be used by SKI. It includes a user-mode application that runs inside the testing VM for the purpose of setting up the kernel and for providing the required test input (e.g., system calls).

The user-mode test framework automatically creates the testing threads/processes, pins each thread/process to a dedicated virtual CPU, issues the hypercalls to mark the beginning of the test (right before the test function is called) and the ending of the test (right after the test function returns), and finally requests the termination of the VM (when all post-test functions have completed). This framework can be used both to manually create test cases (Section 6.3) or to adapt existing test suites to leverage SKI for the interleaving exploration (Section 6.2).

We first implemented the framework targeting Linux and subsequently ported it to FreeBSD, and have been using it to conduct tests on both operating systems. The helper framework itself was easily ported because only few of the system/library calls it relies upon are not part of the POSIX standard (namely the calls to pin threads/processes, which have slightly different interfaces).

5.4 Optimizations and parallelization

In addition to the algorithmic optimizations described in Section 4, we have implemented several other optimizations to improve the performance of SKI. One of our main optimizations avoids resuming from a snapshot for each tested execution, which can take a few seconds in the original version of QEMU. Instead we have implemented in SKI a multi-threaded forking mechanism to take advantage of the copy-on-write semantics offered by the host OS, amortizing the cost of resuming from a snapshot over multiple executions. This benefit is not limited to executions that test the same input because we allow the testing application to receive, through a hypercall, a parameter that specifies the testing input. Thus, from a single snapshot, SKI can explore different inputs and different interleavings, making the overall cost of creating and resuming from a snapshot negligible.

In addition, given that in our testing scenario after each execution we discard most of the state of the VM (e.g. VM RAM and disk contents), we optimized SKI by converting several file system operations, performed by the original QEMU on the host, into memory operations.

Given that our workload is parallelizable, SKI takes advantage of multicore host machines by spawning multiple VMs to perform multiple concurrent tests. We have also implemented a testing infrastructure to distribute the workload across multiple machines, further increasing the testing throughput.

5.5 Bug detectors

Section 3 presented the algorithms and mechanisms that SKI employs to explore the thread interleaving space of the kernel. However, to find concurrency bugs an orthogonal problem needs to be addressed — it is necessary to identify which of these executions triggered bugs.

In Section 6 we show how SKI can be combined with different types of bug detectors — we evaluate SKI using bug detectors to detect crashes, assertion violations, data races and file system inconsistencies. Our implementation detects crashes and assertion violations by monitoring the console output at the VMM level. The detection of data races is also performed at the VMM level by recording racing memory accesses, similarly to DataCollider [32]. File system inconsistencies, in contrast, are detected by running existing file system checkers inside the VM itself after each test.

5.6 Traces and bug diagnosis

To enable the implementation of external bug detectors and to allow the diagnosis of bugs through manual inspection, SKI is able to produce detailed logs of the executions. These traces contain the exact ordering of instructions and the identity of the context responsible for the instructions. In addition, SKI can be configured to produce traces with all the memory accesses and the values of the main CPU registers.

We built some analysis tools that parse these traces to provide useful information. One of our tools produces source code information by disassembling the instructions and by annotating the trace with the source code that generated the instructions (assuming the kernel is compiled with debugging symbols). We also implemented another diagnosis tool that generates the call graph for each execution. While none of these tools is conceptually particularly challenging, in our experience, they complement each other well and make the rich information collected by SKI much more accessible.

Apart from the traces produced by SKI, the bug detectors we built are another important source of diagnostic information. For example, the data race detector that we implemented identifies the exact memory address as well as the instruction addresses involved. As another example, the crash reports produced by the Linux kernel include a detailed stack trace that is very convenient for developers to diagnose bugs.

6 Evaluation

This section evaluates the effectiveness of SKI in revealing real-world kernel concurrency bugs. After describing the configuration that we employed in our experiments, we report our experience in applying SKI to recent and stable versions of the Linux kernel, which resulted in the discovery of several previously unknown concurrency bugs (Section 6.2). We then report on our experiments using SKI to reproduce previously known bugs and comparing it with traditional approaches (Section 6.3).

6.1 Configuration

We conducted our experiments on host machines with dual Intel Xeon X5650 processors and 48 GB of RAM running Linux 3.2.48.1 as the host kernel. To increase the testing throughput, we configured SKI to run 22 testing executions in parallel on each machine and we ran our experiments on up to 12 machines at a time.

For each test case reported in this paper, we configured SKI to use $p = 2$ and we explored 200 schedules in the large-scale experiments to find new bugs (Section 6.2) and $50,000$ schedules in the experiments to reproduce known bugs (Section 6.3). SKI's liveness heuristics used $h_2 = 30$, $h_3 = 20$ and $h_4 = 500,000$ (Section 3.3). We tested several different versions of Linux, ranging from 2.6.28 to 3.13.5, depending on the experiment, and one of the experiments tested FreeBSD, version 8.0. Importantly, the same configuration of SKI was used in all tests: we did not have to modify any settings to adjust SKI to a particular tested kernel version, and we also did not have to modify the kernels under test.

6.2 Finding concurrency bugs

To demonstrate the effectiveness of SKI in finding real world concurrency bugs, we tested several file systems from recent versions of the Linux kernel.

To create the inputs that form the various tests, we modified *fsstress* [44], adding calls to SKI's hypercalls to flag the beginning and the end of the tests, and we modified the test suite to issue concurrent system calls. For convenience we also converted some of the debugging messages to use SKI's own debugging hypercalls. Because one of the file systems (*btrfs*) supports several operations that were not supported by the original *fsstress*, we also added support for twelve of those file system operations (e.g., snapshot/sub-volume operations and dynamic addition/removal of devices). In total, we added or modified 900 lines of code in *fsstress*, of which 700 lines are related to the *btrfs* operations.

6.2.1 Bug detectors

We ran SKI with three bug detectors. The first detector monitors the console output to detect crashes, assertion violations and kernel warning messages. The second detector uses file system checkers (*fsck*), which are specific to each file system and are only supported/mature in the case of some file systems, to detect file system corruption. This bug detector runs inside the VM, in contrast with the others, which are implemented at the VMM level. To limit the performance impact of running *fsck* after each execution, we created small file systems (300 MB) and we mounted the file system in memory using *loop* + *tmpfs* (in addition to the optimizations described in Section 5.4).

The third bug detector consists of a data race detector that we implemented, which analyzes all memory accesses, without sampling. Similarly to other data race detectors [32], our detector finds *racing memory accesses* without distinguishing whether those accesses are performed by synchronization functions. The main chal-

Bug	Kernel	FS	Function	Detector / Failure	E	FS	Status
1	3.11.1	Btrfs	btrfs_find_all_root()	Crash: Null-pointer	41	0.030	Fixed
2	3.11.1	Btrfs	run_clustered_refs()	Crash: Null-pointer + Warning	26	0.020	Fixed
3	3.11.1	Btrfs	record_one_backref()	Warning	74	0.030	Fixed
4	3.11.1	Btrfs	NA	Fsck: Refs. not found	11	0.200	Reported
5	3.12.2+p	Btrfs	btrfs_find_all_root()	Crash: Null pointer	61	0.060	Fixed
6	3.12.2	Btrfs	inode_tree_add()	Warning	53	0.010	Fixed
7	3.13.5	Logfs	indirect_write_alias()	Crash: Null pointer	31	0.065	Reported
8	3.13.5	Logfs	btree_write_alias()	Crash: Invalid paging	142	0.020	Reported
9	3.13.5	Jfs	lbmIODone()	Crash: Assertion	74	0.005	Reported
10	3.13.5	Ext4	ext4_do_update_inode()	Data race	32	0.005	Fixed
11	3.13.5	VFS	generic_fillattr()	Data race	125	0.005	Reported

Table 1: Bugs that have been discovered by SKI in recent versions of the Linux kernel and that we have reported to developers. For the specific input that triggered each bug, we show the number of schedules that were required to expose the bug (E) and the fraction of schedules that triggered the bug (FS). Eventually we found out that bug #3 had previously been reported. A patched version of the kernel, expected to solve bug #1, was tested on request from the developers but SKI revealed that the kernel could still crash in a different location of the same function (bug #5).

		Reports
False data race		76
Data race	Benign	53
	Under investigation	37
	Harmful	24

Table 2: Types of race reports found during our experiments. The numbers displayed refer to the number of reports after associating related races. Note that a single bug may be involved in multiple data races (e.g., if it affects multiple variables).

	Btrfs	Ext4	Jfs	Logfs
SKI	34.7	62.6	61.6	61.2
SKI+ DR	32.1	61.9	59.5	58.8
SKI+ Fsck	6.4	20.8	18.2	N/A
SKI+ Fsck + DR	6.1	20.6	17.9	N/A

Table 3: SKI's throughput (for each machine) with different bug detectors. Throughput is given in thousands of executions per hour. DR denotes the data race detector. Fsck tests on *logfs* are absent due to the lack of compatible mature checkers.

lenge in this case is filtering out the false positives (*false* data races and *benign* data races) [32, 51, 62, 78]. In order to facilitate this manual process, our tool groups together distinct pairs or racing instructions that were found to race directly or transitively. Using this method, we we were able to group together 3114 pairs of races into 190 race reports. Filtering out race reports that were not data races was straightforward, but the difficult part was separating real data races into benign and harmful ones. In some cases, this requires careful analysis of the code and documentation and, ultimately, it may require asking the developers – who may not even agree among themselves. Heuristics could have been used to analyze the results, but unfortunately these typically offer limited help for the more complicated cases. Given this complexity, we gathered some reports (not included in Table 1) that may constitute bugs but are still under analysis, and for which, in some cases, we are still waiting for feedback from the developers. Table 2 shows the number of race reports that we obtained in the file systems tests according to their type.

6.2.2 Results

The results in Figure 1 show that SKI was able to find several unknown concurrency bugs in mature versions of the Linux kernel. One of the bugs found affects the widely used *ext4* file system and six bugs affect the *btrfs* file system – which is expected to soon become the default file system in some distributions [8]. We have reported the 11 bugs listed in Table 1; of those, 6 have already been fixed.

Furthermore, although FS related system calls tend to be expensive, SKI was able to achieve a testing throughput that reached 62 thousand executions per hour on each machine (Table 3). Even though the current performance of SKI proved to be effective, significant performance improvements may still be achievable by using more efficient virtual machines monitors, possibly using hardware acceleration, or even by building SKI using binary

Bug	Kernel	OS Component	Failure	E	FS
A [4]	Linux 2.6.28	Anonymous pipes	Crash	28	0.00572
B [5]	Linux 3.2	Inotify + FAT32	Crash	53	0.13770
C [9]	Linux 3.6.1	Proc file system + Ext4	Semantic	51	0.01004
D [3]	FreeBSD 8.0	Sockets	Semantic	3519	0.00014

Table 4: Known bugs reproduced with SKI. The table shows the number of schedules that were required to expose the bug (E) and the fraction of schedules that triggered the bug (FS). The table shows the kernel version under which we reproduced the bug, the OS components involved and the type of failure that the bug causes.

instrumentation frameworks.

It is worth pointing out that many of the bugs found by SKI are serious – six of the bugs cause the kernel to crash and most of the bugs found cause persistent data loss. For example, the *ext4* bug, which is due to improper synchronization while updating the inodes, causes the field *i_disksize* (containing information about the size of the inodes) to become corrupted. To fix this bug, developers applied patches that involved refactoring the code and the introduction of additional synchronization.

6.3 Reproducing concurrency bugs

We also evaluated the effectiveness of SKI in reproducing previously reported kernel concurrency bugs. To find typical bug reports, we searched the kernel Bugzilla databases, the kernel development histories (i.e., the *git changelogs*), and the mailing list archives. From these sources, we selected four independently confirmed kernel concurrency bugs. We opted for a diverse set of bugs that were particularly well documented. Furthermore, to enable a direct comparison, we considered only bug reports that included instructions for triggering the reported bugs through stress testing.

As listed in Table 4, the selected bugs exhibited different types of failures in various kernel components. Bug A causes a memory access violation (an “Oops” in Linux parlance) in the *pipe* communication mechanism, which can occur during concurrent *open* and *close* calls on anonymous pipes. Bug B also results in a memory access violation and is triggered on some interleavings when a FAT32-formatted partition is unmounted concurrently with the removal of an *inotify* watch[4] associated with the same partition. Bug C does not result in a crash, but rather causes a *read* system call to return corrupted values. Finally, bug D affects FreeBSD and is triggered by concurrent calls on sockets that cause the kernel to incorrectly return error values.

Based on these four bug reports, we determined the system calls that would expose the bugs and produced the corresponding SKI test cases, as described in Section 5.3. For the bugs that had semantic manifestations, i.e., system calls that returned wrong results, we implemented bug-specific detectors, according to the information provided in the bug reports.

SKI exposed bugs A and B by triggering the crash after exploring 28 and 53 schedules, respectively. Bugs C and D were exposed after 51 and 3519 schedules, respectively, causing wrong results to be returned. Given that SKI requires few executions to trigger concurrency bugs, with a suitable test suite (e.g. regression test suites [38]), SKI's throughput is sufficient to reproduce on the order of hundreds of such concurrency bugs per hour (Table 5).

These experiments confirm that SKI is effective at reproducing real-world concurrency bugs. Most importantly, it should be noted that the reproduced bugs stem from two different OS code bases (FreeBSD and Linux) and from a wide range of Linux kernel versions spanning several years of intense development. In fact, even if we ignore the cumulative number of lines changed (i.e., the churn rate) and take into consideration only the increase in the total number of lines of source code, the Linux kernel grew by an impressive 60% from version 2.6.28 (10M SLOC) to version 3.6.1 (16M SLOC). SKI handled the different versions of the Linux kernel and the FreeBSD kernel without requiring any changes to the VMM itself or its configuration, which provides evidence for the considerable versatility intrinsic to SKI's design.

6.3.1 Comparison with stress testing

In the discussions that led to the resolution of these four bugs, the kernel developers proposed non-systematic methods to reproduce them. In particular, they provided simple stress tests, which continuously execute the same operations in a tight loop, waiting until a buggy interleaving occurs. We executed the original stress tests proposed by the developers to compare SKI to a traditional approach. For this purpose, we ran the stress tests in an

[4] Linux's *inotify* interface allows processes to receive change notifications for file system objects such as files, directories, or mount points.

Bug	Throughput
A	302.0
B	169.3
C	218.7
D	501.4

Table 5: SKI's throughput for each machine. Throughput is presented in thousand executions per hour.

unmodified VMM, i.e., without making use of SKI.

Note that without a deep knowledge of the kernel code, in the general case, it is hard to generate stress tests for the bugs that SKI discovered in Section 6.2. The reason for this is that it is not straightforward to ensure that, for every one of the various iterations of the stress test, the state of the kernel is such that it can trigger the concurrency bugs. (SKI avoids this problem because it automatically restores the initial state through snapshotting). Thus, to ensure a more objective comparison between the two approaches, we chose to use stress tests produced by the kernel developers themselves since these are the ones offering better effectiveness guarantees.

As expected, and consistent with earlier comparisons of systematic and unsystematic user-mode concurrency testing approaches [20, 53], SKI proved to be much more effective in reproducing concurrency bugs than the non-systematic approaches. Despite the fact that we gave each stress test up to 24 hours to complete, bug A and bug D were not triggered at all by their corresponding stress tests. While the stress tests for bugs B and C did eventually trigger their corresponding bugs, they required significantly more executions (and time) than SKI: the stress tests required more than $200,000$ iterations (4 hours) to reproduce bug B and more than 800 iterations (1 minute) to trigger bug C, compared to 53 and 51 iterations (both a few seconds), respectively, under SKI.

Overall, the relative difficulty of reproducing bugs with simple stress tests is not surprising given prior comparisons of systematic approaches and stress testing in the context of user-mode applications [20]. Furthermore, this difficulty was also reported by the kernel developers themselves. For example, in the case of bug A (which the stress test failed to reproduce in our experiments) the developer stated that the “failure window is quite small” [6] and recommended introducing a carefully placed sleep statement in the kernel to trigger the bug.

6.3.2 Liveness heuristics

Bug	H1	H2	H3	H4	H*
A	1.72%	0.61%	5.71%	0.57%	7.97%
B	88.80%	49.70%	0.05%	13.73%	88.93%
C	1.50%	23.56%	0.00%	0.00%	25.06%
D	0.53%	2.66%	0.00%	0.00%	3.05%

Table 6: Percentage of schedules that triggered the liveness heuristics. H* refers to the percentage of schedules that trigger any heuristic.

Bug	H1	H2	H3	H4	H*
A	0.08	0.01	0.06	0.01	0.17
B	14.97	1.59	36.38	0.14	53.08
C	0.01	0.44	0.00	0.00	0.45
D	0.01	0.03	0.00	0.00	0.03

Table 7: Average number of times the liveness heuristics were triggered per schedule. H* refers to the percentage of schedules that trigger any heuristic.

We instrumented SKI to log the activation of SKI's heuristics. Using this data we calculated the percentage of schedules that triggered each of the heuristics (Table 6) and the average number of times each heuristic was triggered per schedule (Table 7).

The results show that some of the schedules do not trigger heuristics. This is expected to happen when SKI chooses schedules in which threads do not experience lock contention and is more likely to occur in operating systems that are well optimized for scalability. Even though not all of the tests activate all heuristics, all heuristics were activated in at least one of the test cases.

In addition, we observed that in these tests the heuristics were triggered at 167 distinct instruction addresses. The large number of distinct addresses is indicative of the challenges that would result from manually annotating the kernel to infer thread liveness, as opposed to relying on the four simple heuristics implemented by SKI.

6.3.3 Effectiveness of communication points

To evaluate the effectiveness of the optimization of keeping track of communication points and allowing reschedules to occur only at these points (described in Section 4), we calculated for each test case the average number of instructions and the average number of communication points executed per run. As shown in Table 8, this optimization reduced the number of potential reschedule points by up to an order of magnitude in our experiments, thereby avoiding the wasteful exploration of redundant, effectively equivalent schedules. This shows the importance of this optimization to the scalability of SKI.

Bug	I	CP	I/CP
A	87673.5	12511.2	7.00
B	210693.0	23432.8	8.99
C	65126.9	6372.3	10.22
D	22641.3	6503.2	3.48

Table 8: Effectiveness of the communication points optimization described in Section 4. The table shows for each reproduced bug the average number of instructions executed per run (I) and the average communication points executed per run (CP). The last column characterizes the optimization's effectiveness as the ratio of the two metrics.

7 Discussion

SKI proposes a VMM-based scheduler. In this section, we discuss some of the implications of this choice.

A limitation of relying on a VMM is that the kernel running inside a virtual machine is limited to using the hardware virtualized by the VMM. For a testing tool, it means that it is not possible to reproduce bugs that require hardware that is not virtualized by the VMM. However, we believe this does not detract significantly from SKI's practical value because the size of the device-independent kernel core is already considerable. Further, it may be possible to overcome the VMM dependency by building an equivalent tool based on kernel binary instrumentation, which is an active area of research [33].

The choice of a VMM-based approach has another important consequence. Because the VMM emulates one instruction at a time, and propagates its effects to all other CPUs immediately afterwards, concurrency bugs that arise from wrongly assuming a strong memory model are not necessarily exposed. This is because some CPUs offer weaker memory models, which can have very complex semantics, to the point where official specifications have been found to not match the observed semantics of hardware [11]. This is a complex problem — significant effort has been directed at simply studying the semantics of CPUs with relaxed memory models [61] — and we believe that effectively diagnosing this type of concurrency bug will likely require more specialized tools. Such bugs are currently not the target of SKI.

8 Related Work

Schedule space exploration. The traditional way to test applications for concurrency bugs relies on manually created stress tests. To increase the chances of unmasking concurrency bugs, researchers have proposed various tools that rely on introducing sleeps in the code to disrupt the scheduling of threads [14, 19, 32, 56, 63, 64]. The common limitation to these approaches is that they do not systematically explore the thread interleaving space.

To address these limitations, a different class of tools has been proposed to test for concurrency bugs [20, 53, 54]. This approach relies on taking full control of the scheduling of threads to avoid redundant interleavings and therefore increases the effectiveness of testing [20]. A previous attempt [17] to systematically test kernel code has focused on small-scale educational kernels and relied on modifications to the tested kernels. SKI follows the systematic approach, but distinguishes itself from existing tools by being applicable to kernel code and by being scalable to real-world kernels.

Because the schedule space is extremely large, systematic tools take advantage of different techniques to restrict the schedule exploration while still ensuring effectiveness. Examples used in the context of finding user-mode concurrency bugs include preemption bounding [53], reschedule bounding [20] and the elimination of redundant schedules [36]. Other work has proposed limiting the valid run-time schedules by reducing or eliminating the schedule non-determinism [27–30, 46, 71, 75]. Restricting the kernel schedules by applying these techniques could further increase the effectiveness of SKI.

Symbolic execution [21, 25] is an analysis technique that systematically explores the application execution path space by keeping track, during execution, of symbolic values instead of concrete values. Symbolic execution has been applied to multi-threaded applications by implementing a custom user-mode scheduler [41]. More recently, SymDrive [57] has been successful at testing kernel device drivers using symbolic execution, although it requires modifications to the kernel and does not target concurrency bugs. Similarly, SWIFT [22] uses symbolic execution to test kernel file system checkers but does not target concurrency bugs. By using SKI's ability to instrument kernel schedules, it may be possible to leverage the symbolic execution approach in the context of testing the kernel for concurrency bugs.

Similarly to shared-memory systems, which are the focus of SKI, distributed systems are also prone to schedule-dependent bugs [45, 47, 58] and the complexity of distributed systems also requires dedicated techniques to scale to real-world applications. For example, CrystalBall [73] proposes model checking live systems and steering their execution away from states that trigger bugs. By exploring states based on snapshots of live systems, CrystalBall is able to explore states that are more likely to be relevant to the current execution than conducting the entire exploration from a single initial state. MoDist [74] also finds bugs in distributed systems but does so transparently, without requiring implementations to be written in special languages. MoDist is able to scale to complex implementations by judiciously simulating events that typically trigger bugs, such as the reordering of messages and the expiration of timers.

Detecting concurrency bugs. Different types of bug detectors have been proposed to detect at runtime whether an execution is anomalous [2, 18, 32, 34, 35, 49, 51, 62, 68, 78]. Detecting anomalous executions is a challenge complementary to the exploration of the interleaving space. DataCollider [32] is a kernel data race detector that randomly pauses CPUs, through the use of hardware breakpoints, to cause non-systematic schedule diversity. In Section 5.5 we show how we combined DataCollider's data race detection mechanism with SKI to detect data races. RedFlag [66] is another example of a concurrency bug detector for the kernel that combines a block-based atomicity checker with a lockset based data race detector. In Section 5.5, we describe in detail how SKI leverages various bug detectors.

Deterministic replay. Determinism is valuable for diagnosing concurrency bugs [13, 15, 27, 28, 30, 46], but ensuring determinism is orthogonal to the systematic interleaving exploration problem. Given the same, fixed input parameters, SKI, like its user-mode counterparts, can deterministically re-execute the same schedule, provided the kernel is given identical input in each run. Currently, SKI does not ensure that the same hardware input is provided to the kernel (e.g. low-granularity timer values). Input determinism could be achieved through the use of another layer running below the VMM [46] or by modifying the VMM [31, 72].

Input space exploration. Dynamic testing techniques require running the tested software and providing it with testing input. The traditional approach has relied on manually writing test cases [38], but more sophisticated approaches have been proposed to address this challenge. Such approaches include blackbox fuzzers [16], semantically-aware fuzzers [1, 10] and symbolic execution techniques [21, 37, 42]. Because file systems have a particularly large input space and are critical components in the system, file system testing has been a particularly active area of research [12, 22, 50, 76, 77]. Even though the focus of SKI is on the exploration of the interleaving space, to evaluate SKI we explored the kernel input space with an existing file system test suite, *fsstress* [44].

Virtual machine introspection (VMI). Several VMM mechanisms have been proposed to infer high-level information of virtual machines [23]. In many cases the purpose of these mechanisms is to increase performance. Examples include improving the host memory usage by inferring which guest memory is actively being used [24], improving IO performance by anticipating IO requests [40] and improving the scalability of virtual machine monitors by inferring whether the virtual machine is executing critical sections [65, 70]. In addition, VMI techniques have been leveraged to gather information about virtual machines in security contexts [55]. Using the introspection approach, SKI infers the liveness of threads for the purpose of achieving fine-level control over the threads schedules. For example, SKI leverages the observation that the PAUSE instruction is typically associated with spin-locks, as does the work of Wang et. al [70] in the context of increasing VMM performance.

9 Conclusion

This paper introduces SKI, the first practical testing tool to systematically explore the interleaving space of real-world kernel code. SKI does not require any modifications to tested kernels, nor does it require knowledge of the semantics of any kernel synchronization primitives. We detailed key optimizations that make SKI scale to real-world code, and we have shown that SKI is effective at finding buggy schedules in both FreeBSD and various versions of the Linux kernel, without changing or annotating the tested kernel.

As future work, we plan to explore different bug detectors and to leverage the control provided by SKI to effectively explore the input space.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback and our shepherd, Junfeng Yang, for his help. Pedro Fonseca was partially supported by a fellowship from the Portuguese Foundation for Science and Technology (FCT). Rodrigo Rodrigues was funded by the European Research Council under an ERC starting grant.

References

[1] A Linux System call fuzz tester. http://codemonkey.org.uk/projects/trinity/.

[2] ANNOUNCE: Lock validator. http://lwn.net/Articles/185605/.

[3] Bug 144061 - [socket] race on unix socket close. https://bugs.freebsd.org/bugzilla/show_bug.cgi?id=144061.

[4] Bug 14416 - Null pointer dereference in fs/pipe.c . http://bugzilla.kernel.org/show_bug.cgi?id=14416.

[5] Bug 22602 - Oops while unmounting an USB key with a FAT filesystem. https://bugzilla.kernel.org/show_bug.cgi?id=22602.

[6] FS: pipe.c null pointer dereference. https://git.kernel.org/cgit/linux/kernel/git/stable/stable-queue.git/tree/queue-2.6.31/fs-pipe.c-null-pointer-dereference.patch?id=36e97dec52821f76536a25b763e320eb7434c2a5.

[7] Kernel threads made easy. http://lwn.net/Articles/65178/.

[8] OpenSUSE News. https://news.opensuse.org/2014/03/19/development-for-13-2-kicks-off/.

[9] Patch ”ext4: fix crash when accessing /proc/mounts concurrently” has been added to the 3.6-stable tree. http://www.mail-archive.com/stable@vger.kernel.org/msg19380.html.

[10] Aitel, D. The advantages of block-based protocol analysis for security testing. Tech. rep., Immunity, Inc., 2002.

[11] Alglave, J., Fox, A., Ishtiaq, S., Myreen, M. O., Sarkar, S., Sewell, P., and Nardelli, F. Z. The semantics of POWER and ARM multiprocessor machine code. In *Proc. of Workshop on Declarative Aspects of Multicore Programming (DAMP)* (2008).

[12] Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., Bairavasundaram, L. N., Denehy, T. E., Popovici, F. I., Prabhakaran, V., and Sivathanu, M. Semantically-smart disk systems: Past, present, and future. *SIGMETRICS Perform. Eval. Rev. 33*, 4 (Mar. 2006), 29–35.

[13] Aviram, A., Weng, S.-C., Hu, S., and Ford, B. Efficient system-enforced deterministic parallelism. In *Proc. of Operating System Design and Implementation (OSDI)* (2010).

[14] Ben-Asher, Y., Eytani, Y., Farchi, E., and Ur, S. Noise makers need to know where to be silent – Producing schedules that find bugs. In *Proc. of International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)* (2006).

[15] Berger, E. D., Yang, T., Liu, T., and Novark, G. Grace: Safe multithreaded programming for C/C++. In *Proc. of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2009).

[16] Bird, D. L., and Munoz, C. U. Automatic generation of random self-checking test cases. *IBM Syst. J. 22*, 3 (Sept. 1983), 229–245.

[17] Blum, B. *Landslide: Systematic Dynamic Race Detection in Kernel Space*. MS thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, May 2012. http://www.pdl.cmu.edu/PDL-FTP/associated/CMU-CS-12-118.pdf.

[18] Bond, M. D., Coons, K. E., and McKinley, K. S. PACER: Proportional detection of data races. In *Proc. of Programming Languages Design and Implementation (PLDI)* (2010).

[19] Bron, A., Farchi, E., Magid, Y., Nir, Y., and Ur, S. Applications of synchronization coverage. In *Proc. of Symposium on Principles and Practice of Parallel Programming (PPoPP)* (2005).

[20] Burckhardt, S., Kothari, P., Musuvathi, M., and Nagarakatte, S. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2010).

[21] Cadar, C., Dunbar, D., and Engler, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of Operating System Design and Implementation (OSDI)* (2008).

[22] Carreira, J. a., Rodrigues, R., Candea, G., and Majumdar, R. Scalable testing of file system checkers. In *Proc. of European Conference on Computer Systems (EuroSys)* (2012).

[23] Chen, P. M., and Noble, B. D. When virtual is better than real. In *Proc. of Hot Topics in Operating Systems (HotOS)* (2001).

[24] Chiang, J.-H., Li, H.-L., and Chiueh, T.-c. Introspection-based memory de-duplication and migration. In *Proc. of International Conference on Virtual Execution Environments (VEE)* (2013).

[25] Chipounov, V., Kuznetsov, V., and Candea, G. S2E: A platform for in-vivo multi-path analysis of software systems. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2011).

[26] Clements, A. T., Kaashoek, M. F., and Zeldovich, N. RadixVM: Scalable address spaces for multithreaded applications. In *Proc. of European Conference on Computer Systems (EuroSys)* (2013).

[27] Cui, H., Simsa, J., Lin, Y.-H., Li, H., Blum, B., Xu, X., Yang, J., Gibson, G. A., and Bryant, R. E. Parrot: A practical runtime for deterministic, stable, and reliable threads. In *Proc. of Symposium on Operating System Principles (SOSP)* (2013).

[28] Cui, H., Wu, J., Gallagher, J., Guo, H., and Yang, J. Efficient deterministic multithreading through schedule relaxation. In *Proc. of Symposium on Operating System Principles (SOSP)* (2011).

[29] Cui, H., Wu, J., Tsai, C.-C., and Yang, J. Stable deterministic multithreading through schedule memoization. In *Proc. of Operating System Design and Implementation (OSDI)* (2010).

[30] Devietti, J., Lucia, B., Ceze, L., and Oskin, M. DMP: Deterministic shared memory multiprocessing. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2009).

[31] Dunlap, G. W., Lucchetti, D. G., Fetterman, M. A., and Chen, P. M. Execution replay of multiprocessor virtual machines. In *Proc. of International Conference on Virtual Execution Environments (VEE)* (2008).

[32] Erickson, J., Musuvathi, M., Burckhardt, S., and Olynyk, K. Effective data-race detection for the kernel. In *Proc. of Operating System Design and Implementation (OSDI)* (2010).

[33] FEINER, P., BROWN, A. D., AND GOEL, A. Comprehensive kernel instrumentation via dynamic binary translation. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2012).

[34] FLANAGAN, C., AND FREUND, S. N. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proc. of Symposium on Principles of Programming Languages (POPL)* (2004).

[35] FONSECA, P., LI, C., AND RODRIGUES, R. Finding complex concurrency bugs in large multi-threaded applications. In *Proc. of European Conference on Computer Systems (EuroSys)* (2011).

[36] GODEFROID, P. Model checking for programming languages using VeriSoft. In *Proc. of Symposium on Principles of Programming Languages (POPL)* (1997).

[37] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: Directed automated random testing. *SIGPLAN Not. 40*, 6 (2005), 213–223.

[38] GRAVES, T. L., HARROLD, M. J., KIM, J.-M., PORTER, A., AND ROTHERMEL, G. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol. 10*, 2 (Apr. 2001), 184–208.

[39] HOLZMANN, G. J. Mars code. *Commun. ACM 57*, 2 (2014), 64–73.

[40] JONES, S. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Antfarm: Tracking processes in a virtual machine environment. In *Proc. of Annual Technical Conference (ATC)* (2006).

[41] KASIKCI, B., ZAMFIR, C., AND CANDEA, G. Data races vs. data race bugs: Telling the difference with portend. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2012).

[42] KING, J. C. Symbolic execution and program testing. *Commun. ACM 19*, 7 (July 1976), 385–394.

[43] KOEHNEMANN, H., AND LINDQUIST, T. E. Towards target-level testing and debugging tools for embedded software. In *TRI-Ada* (1993).

[44] LARSON, P. Testing linux with linux test project. In *Proc. of Ottawa Linux Symposium (OLS)* (2002).

[45] LI, S., ZHOU, H., LIN, H., XIAO, T., LIN, H., LIN, W., AND XIE, T. A characteristic study on failures of production distributed data-parallel programs. In *Proc. of International Conference on Software Engineering (ICSE)* (2013).

[46] LIU, T., CURTSINGER, C., AND BERGER, E. D. Dthreads: Efficient deterministic multithreading. In *Proc. of Symposium on Operating System Principles (SOSP)* (2011).

[47] LIU, X. WiDS checker: Combating bugs in distributed systems. In *Proc. of Networked Systems Design and Implementation (NSDI)* (2007).

[48] LU, S., PARK, S., SEO, E., AND ZHOU, Y. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2008).

[49] LU, S., TUCEK, J., QIN, F., AND ZHOU, Y. AVIO: detecting atomicity violations via access interleaving invariants. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2006).

[50] MA, A., DRAGGA, C., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Ffsck: The fast file system checker.

[51] MARINO, D., MUSUVATHI, M., AND NARAYANASAMY, S. LiteRace: Effective sampling for lightweight data-race detection. In *Proc. of Programming Languages Design and Implementation (PLDI)* (2009).

[52] MCKENNEY, P. E., AND SLINGWINE, J. D. Read-copy update: Using execution history to solve concurrency problems. In *Proc. of International Conference on Parallel and Distributed Computing and Systems (PDCS)* (1998).

[53] MUSUVATHI, M., QADEER, S., BALL, T., BASLER, G., NAINAR, P. A., AND NEAMTIU, I. Finding and reproducing heisenbugs in concurrent programs. In *Proc. of Operating System Design and Implementation (OSDI)* (2008).

[54] NAGARAKATTE, S., BURCKHARDT, S., MARTIN, M. M., AND MUSUVATHI, M. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *Proc. of Programming Languages Design and Implementation (PLDI)* (2012).

[55] NANCE, K., BISHOP, M., AND HAY, B. Virtual machine introspection: Observation or interference? *IEEE Security and Privacy 6*, 5 (Sept. 2008), 32–37.

[56] PARK, S., LU, S., AND ZHOU, Y. CTrigger: Exposing atomicity violation bugs from their hiding places. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2009).

[57] RENZELMANN, M. J., KADAV, A., AND SWIFT, M. M. SymDrive: Testing drivers without devices. In *Proc. of Operating System Design and Implementation (OSDI)* (2012).

[58] REYNOLDS, P., KILLIAN, C., WIENER, J. L., MOGUL, J. C., SHAH, M. A., AND VAHDAT, A. Pip: Detecting the unexpected in distributed systems. In *Proc. of Networked Systems Design and Implementation (NSDI)* (2006).

[59] RUSSELL, P. R. Unreliable Guide To Locking. `http://kernelbook.sourceforge.net/kernel-locking.pdf`.

[60] SAHOO, S. K., CRISWELL, J., AND ADVE, V. S. An empirical study of reported bugs in server software with implications for automated bug diagnosis. Tech. Report 2142/13697, University of Illinois, 2009.

[61] SARKAR, S., SEWELL, P., NARDELLI, F. Z., OWENS, S., RIDGE, T., BRAIBANT, T., MYREEN, M. O., AND ALGLAVE, J. The semantics of x86-CC multiprocessor machine code. In *Proc. of Symposium on Principles of Programming Languages (POPL)* (2009).

[62] SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. Eraser: A dynamic data race detector for multi-threaded programs. In *Proc. of Symposium on Operating System Principles (SOSP)* (1997).

[63] SEN, K. Race directed random testing of concurrent programs. In *Proc. of Programming Languages Design and Implementation (PLDI)* (2008).

[64] STOLLER, S. D. Testing concurrent Java programs using randomized scheduling. In *Proc. of Workshop on Runtime Verification (RV)* (2002).

[65] UHLIG, V., LEVASSEUR, J., SKOGLUND, E., AND DANNOWSKI, U. Towards scalable multiprocessor virtual machines. In *Proc. of Conference on Virtual Machine Research And Technology Symposium (VM)* (2004).

[66] URTEAGA, I. N., BARNHART, K., AND HAN, Q. REDFLAG: A run-time, distributed, flexible, lightweight, and generic fault detection service for data-driven wireless sensor applications. *Pervasive Mob. Comput. 5* (October 2009), 432–446.

[67] VALOIS, J. D. Implementing lock-free queues. In *Proc. of International Conference on Parallel and Distributed Computing and Systems (PDCS)* (1994).

[68] VEERARAGHAVAN, K., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. Detecting and surviving data races using complementary schedules. In *Proc. of Symposium on Operating System Principles (SOSP)* (2011).

[69] WAGNER, S., JÜRJENS, J., KOLLER, C., AND TRISCHBERGER, P. Comparing bug finding tools with reviews and tests. In *Proceedings of the 17th IFIP TC6/WG 6.1 International Conference on Testing of Communicating Systems (Testcom)* (2005).

[70] WANG, Z., LIU, R., CHEN, Y., WU, X., CHEN, H., ZHANG, W., AND ZANG, B. COREMU: A scalable and portable parallel full-system emulator. In *Proc. of Symposium on Principles and Practice of Parallel Programming (PPoPP)* (2011).

[71] WU, J., TANG, Y., HU, G., CUI, H., AND YANG, J. Sound and precise analysis of parallel programs through schedule specialization. In *Proc. of Programming Languages Design and Implementation (PLDI)* (2012).

[72] XU, M., MALYUGIN, V., SHELDON, J., VENKITACHALAM, G., AND WEISSMAN, B. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Proc. of Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)* (2007).

[73] YABANDEH, M., KNEZEVIC, N., KOSTIC, D., AND KUNCAK, V. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *Proc. of Networked Systems Design and Implementation (NSDI)* (2009).

[74] YANG, J., CHEN, T., WU, M., XU, Z., LIU, X., LIN, H., YANG, M., LONG, F., ZHANG, L., AND ZHOU, L. MODIST: Transparent model checking of unmodified distributed systems. In *Proc. of Networked Systems Design and Implementation (NSDI)* (2009).

[75] YANG, J., CUI, H., WU, J., TANG, Y., AND HU, G. Determinism is not enough: Making parallel programs reliable with stable multithreading. *Communications of the ACM* (2014).

[76] YANG, J., SAR, C., AND ENGLER, D. EXPLODE: a lightweight, general system for finding serious storage system errors. In *Proc. of Operating System Design and Implementation (OSDI)* (2006).

[77] YANG, J., TWOHEY, P., ENGLER, D., AND MUSUVATHI, M. Using model checking to find serious file system errors. In *Proc. of Operating System Design and Implementation (OSDI)* (2004).

[78] YU, Y., RODEHEFFER, T., AND CHEN, W. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *Proc. of Symposium on Operating System Principles (SOSP)* (2005).