Reverse Debugging of Kernel Failures in Deployed Systems

Xinyang Ge, *Microsoft Research;* Ben Niu, *Microsoft;* Weidong Cui, *Microsoft Research*

https://www.usenix.org/conference/atc20/presentation/ge

This paper is included in the Proceedings of the
2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

Reverse Debugging of Kernel Failures in Deployed Systems

Xinyang Ge
Microsoft Research

Ben Niu
Microsoft

Weidong Cui
Microsoft Research

Abstract

Post-mortem diagnosis of kernel failures is crucial for operating system vendors because kernel failures impact the reliability and security of the whole system. However, debugging kernel failures in deployed systems remains a challenge because developers have to speculate the conditions leading to the failure based on limited information such as memory dumps. In this paper, we present Kernel REPT, the first practical reverse debugging solution for kernel failures that is highly efficient, imposes small memory footprint and requires no extra software layer. To meet this goal, Kernel REPT employs efficient hardware tracing to record the kernel's control flow on each processor, recognizes the control flow of each software thread based on the context switch history, and recovers its data flow by emulating machine instructions and hardware events such as interrupts and exceptions. We design, implement, and deploy Kernel REPT on Microsoft Windows. We show that developers can use Kernel REPT to do interactive reverse debugging and find the root cause of real-world kernel failures. Kernel REPT also enables automatic root-cause analysis for certain kernel failures that were hard to debug even manually. Furthermore, Kernel REPT can proactively identify kernel bugs by checking the reconstructed execution history against a set of predetermined invariants.

1 Introduction

Post-mortem diagnosis of software failures in deployed systems is becoming increasingly important for today's software development process. Many software vendors such as Microsoft and Apple have insider programs to test their latest software before it is released to the general public. Software developers rely more and more on debugging failures reported from early adopters to fix critical issues before every software release. In particular, the operating system is of the utmost importance because it is the foundation of the software stack and its bugs can have catastrophic impact on the reliability and security of the whole system.

Debugging kernel failures in deployed systems has been a challenge. The fundamental reason is that developers have to speculate on the conditions leading up to the failure based on the limited information available for post-mortem diagnosis such as crashing stacks or memory dumps. The complexity of the kernel makes developers' speculation ineffective in many cases. For example, the Windows kernel checks a set of invariants upon returning to the user mode, and terminates the system if any invariant is violated. Such a failure leaves developers an empty call stack, which makes it almost impossible to debug.

This motivates us to build a *practical* solution that enables developers to go back in time and examine the root cause of kernel failures in deployed systems. Reverse debugging is not a new idea [5, 10], and researchers and practitioners have developed record and replay solutions that can precisely log the execution of the whole system [16, 21, 22, 28]. However, existing whole-system record and replay solutions require the target operating system to run on top of emulation or virtualization, use an excessive amount of memory and storage space to support the record and replay, and introduce significant performance slowdown. On contrary, a practical reverse debugging solution for deployed systems must be able to provide a high-fidelity execution history for post-mortem diagnosis while meeting the requirements of low performance overhead, small memory footprint, no additional setup of software emulation or virtualization, minimal change to the operating system, and zero compromise on the backward compatibility with existing applications.

In this paper, we present Kernel REPT, the first practical solution for reverse debugging of kernel failures in deployed systems. It is an extension of REPT [19], a reverse debugging solution for user-mode applications. Kernel REPT leverages online hardware tracing to log the control flow of kernel executions and performs an offline binary analysis to recover the data flow. By configuring the hardware to trace the target kernel inside the kernel itself, Kernel REPT avoids the extra layer of software emulation or virtualization, has the minimal change to the target operating system, and is fully compatible with existing applications. Furthermore, hardware tracing is shown to be efficient [27].

Kernel REPT traces the kernel execution on each CPU core instead of on each software thread as done in REPT. This helps Kernel REPT achieve a small memory footprint because the number of CPU cores is much less than the number of software threads in a system. In this tracing configuration, one trace buffer may contain traces of multiple threads and the trace of one thread may span multiple trace buffers. To allow reverse debugging over the execution of a thread, Kernel REPT requires the context switch history to assemble the trace of the thread. However, Kernel REPT cannot infer the context switch history based on the control flow or the memory dump. Instead, Kernel REPT logs the context switch events during runtime and uses them to reconstruct the execution of a given thread during offline analysis. This way Kernel REPT can provide reverse debugging over the execution of a thread on top of the core-based tracing.

REPT performs forward and backward instruction emulation to recover the program's data flow, however, it is insufficient to just emulate the semantics of machine instructions in Kernel REPT. This is because a processor modifies the kernel state when a hardware event such as interrupts or exceptions occurs. To correctly recover the kernel state, Kernel REPT needs to emulate the semantics of these hardware events properly. However, these hardware events are not explicitly logged in the control flow trace, and different events may make different changes to the kernel state. Kernel REPT solves this problem by leveraging the kernel configuration of hardware event handlers. For instance, Kernel REPT can tell a page fault just happened when the page fault handler is executed as shown in the control flow trace.

We implement Kernel REPT and deploy it on a billion devices running Microsoft Windows. Our experiments show that Kernel REPT is efficient as it incurs less than 10% slowdown for microbenchmarks and 2% slowdown for applications like Nginx and Chrome. Windows kernel developers use Kernel REPT to debug real-world kernel failures and find the root cause of some kernel bugs that have existed for a decade and caused innumerable failures.

The usage of Kernel REPT is not limited to interactive reverse debugging. To this end, we develop an automatic root-cause analysis for a common class of kernel failures where the kernel fails when it detects that certain resources are not properly released before returning to the user mode. This class of failures is hard to debug even manually without Kernel REPT because the kernel stack is empty when a failure happens. Based on the execution history reconstructed by Kernel REPT, our analysis can *automatically* identify the buggy function for 136 out of 188 real-world kernel failures of this class. This automatic root-cause analysis is deployed as part of Microsoft's error reporting service [24].

The reconstructed kernel execution history can enable not only automatic root-cause analysis but also proactive bug detection. A common kernel bug pattern is that the *exception* handling code fails to properly release resources acquired during the execution wrapped in the *try* block. We build a hybrid analysis to proactively look for this bug pattern by dynamically analyzing the execution in a *try* block and statically analyzing the code in the *exception* handling block. By analyzing thousands of real-world kernel execution histories, our hybrid analysis finds 17 new bugs in the Windows kernel, and all of them are confirmed and fixed.

2 Overview

The goal of Kernel REPT is to reconstruct the execution history of kernel failures in deployed systems for effective postmortem diagnosis without incurring noticeable runtime overhead. As an extension of REPT [19], it utilizes hardware tracing to log the kernel's control flow to a circular buffer at runtime, and recovers its data flow by running binary analysis on the recorded control flow and the memory dump of a failure. In the rest of this section, we first provide a background of REPT. Then we discuss the challenges faced by Kernel REPT.

2.1 REPT

REPT shows a promising way to do reverse debugging for user-mode failures in deployed systems. REPT logs a program's control flow into a *per-thread* circular buffer with low runtime overhead via hardware tracing (e.g., Intel Processor Trace [18]), and then recovers its data flow offline by combining the control flow with the memory dump taken at the failure point. To do so, REPT runs an iterative binary analysis that performs forward and backward instruction emulation on the recorded instruction sequence to infer the program state before every instruction based on the final state stored in the memory dump. REPT checks for conflicts during the execution history reconstruction and performs error corrections based on heuristics. REPT also supports multithreaded programs by merging the instruction sequences of different threads into a partially ordered single instruction sequence based on the fine-grained timestamps logged by the hardware tracing, and limiting the use of concurrent shared memory writes in the binary analysis if their exact order cannot be determined. The reconstructed execution history is not perfect, but it is shown that REPT achieves high accuracy and enables effective reverse debugging of real-world application failures.

To illustrate how REPT works, we show an example borrowed from the REPT paper [19, Figure 2] in Figure 1. This example has 5 instructions in the control-flow trace ($I_1..I_5$). The program state S_i represents the state *after* the execution of instruction I_i. Therefore, the final program state S_5 stored in the memory dump has `rax`=3, `rbx`=0, and `[g]`=3. REPT performs backward and forward analysis iteratively

			Iteration 1	Iteration 2	Iteration 3
		S_0	↑ {rax=?, rbx=?, [g]=3} →		↑ {rax=?, rbx=?, [g]=2}
I_1	`lea rbx, [g]`	S_1	↑ {rax=?, rbx=?, [g]=3}	↓ {rax=?, rbx=g, [g]=3}	↑ {rax=?, rbx=g, [g]=2}
I_2	`mov rax, 1`	S_2	↑ {rax=?, rbx=?, [g]=3}	↓ {rax=1, rbx=g, [g]=3}	↑ {rax=1, rbx=g, **[g]=2**}
I_3	`add rax, [rbx]`	S_3	↑ {rax=3, rbx=?, **[g]=3**}	↓ {**rax=3**, rbx=g, [g]=3}	↑ {rax=3, rbx=g, **[g]=?**}
I_4	`mov [rbx], rax`	S_4	↑ {rax=3, rbx=?, [g]=3}	↓ {rax=3, rbx=g, [g]=3}	↑ {rax=3, rbx=g, [g]=3}
I_5	`xor rbx, rbx`	S_5	↑ {rax=3, rbx=0, [g]=3}	↓ {rax=3, rbx=0, [g]=3} →	

Figure 1: "This example shows how REPT's iterative analysis recovers register and memory values when there exist irreversible instructions with memory accesses. We use "?" to represent "unknown", and use "g" to represent the memory address of a global variable. Some values are in bold-face because they represent key updates in the analysis. We skip the fourth iteration which will recover [g]'s value to be 2 due to the space constraint." [19, Figure 2]

to recover data values. In the first iteration, REPT does not update the global variable `[g]` in S_3 because `rbx`'s value is unknown. In the second iteration, there is a conflict for `rax`'s value in S_3. Its original value is 3, but the forward analysis would infer value 4 for it (`rax` + `[g]` = 1 + 3 = 4). REPT keeps the original value of 3 because it is from the final program state stored in the memory dump. In the third iteration, REPT recovers `[g]`'s value to be 2 based on `rax`'s value before and after the `add` instruction I_3.

For the purpose of this paper, we abstract REPT as a mechanism that takes as the input a final machine state and its preceding instruction sequence, and outputs the recovered machine state before every instruction with high accuracy. Kernel REPT leverages this data recovery mechanism to enable reverse debugging of kernel failures.

2.2 Challenges

A straw-man solution to support reverse debugging of kernel failures is to modify REPT to trace the kernel execution of each software thread and run the same binary analysis on a kernel memory dump. However, this simple solution does not work for two reasons.

First, it incurs unacceptable memory overhead. The kernel is shared by all threads on a system and allocating a trace buffer for each of them can consume an unpredictable amount of memory, especially when a system can have thousands or even tens of thousands of threads.

Second, the kernel has to handle hardware events of which user-mode applications are unaware. For instance, interrupts and exceptions can change the kernel's stack layout without executing any explicit instruction. The details of these hardware events such as the exception vector are not logged by the hardware tracing. However, such information is important for the data flow recovery because different types of hardware events have different architectural effects that must be emulated.

3 Kernel REPT

In this section, we present the design of Kernel REPT. We first describe how Kernel REPT avoids excessive memory consumption via per-core tracing while still allowing reverse debugging over the execution of a thread. Then we present how Kernel REPT handles hardware events when performing the offline binary analysis to recover the data flow.

3.1 Per-Core Tracing

To minimize the memory footprint, Kernel REPT chooses to do *per-core tracing* instead of per-thread tracing for the kernel. That is, Kernel REPT allocates a circular trace buffer for each logical core and logs the kernel-mode execution on a core to its corresponding buffer. Kernel REPT does not log the control flow of user-mode executions because the actual machine code executed in the user mode is not directly related to the root cause of kernel failures. Per-core tracing ensures that Kernel REPT's memory usage is linear in the number of logical cores on a system, and the number of logical cores is fixed and small compared to the number of software threads. This allows Kernel REPT to configure a large trace buffer for each core when necessary without the risk of exhausting the memory.

Per-core tracing does come with its own problems. It is more intuitive for developers to follow the execution on a software thread as opposed to a hardware core. On a multiprocessor system, per-core tracing may mix traces of different threads into one trace buffer and spread the trace of a single thread into multiple trace buffers. This requires Kernel REPT to obtain the context switch history to identify the trace of a single thread.

Ideally, Kernel REPT should recover the context switch history from a per-core trace by leveraging the binary analysis. However, the binary analysis cannot effectively reverse the context switch routine because the scheduling history is neither preserved in the memory dump nor can be inferred from the recorded instruction sequence. We show the pseudo code of a typical context switch routine in Figure 2. The context switch routine saves the register context of the previous

```
; pseudo code for context switch
; rdi points to the old thread
; rsi points to the new thread
push rax                    ; save GPRs
push rbx
...
mov KernelStack[rdi], rsp ; switch stack
mov rsp, KernelStack[rsi]
...                         ; restore GPRs
pop rbx
pop rax
ret
```

Figure 2: Pseudo code for context switch. Basically, the context switch routine saves the register context to the previous thread's stack and the stack pointer to the previous thread's internal data structure, and then restores the context of the new thread by doing the opposite operations.

Type	Origin	Details
Interrupt	User/Kernel	Vector number
Exception	User/Kernel	Vector number
Syscall	User	N/A

Table 1: Information about hardware events needed for software emulation.

thread on its stack, swaps the stack pointer, and then restores the register context of the newly scheduled thread to resume its execution. In this process, the context switch routine does not save the information about the previous thread before resuming the execution of the new thread, and hence the binary analysis is unable to recover the scheduling history. This also makes the binary analysis ineffective when applied directly to the per-core trace because the register values before a context switch cannot be recovered. Therefore, Kernel REPT chooses to log the context switch history in software.

3.2 Handling Hardware Events

The operating system manages hardware resources and has to handle hardware events. Therefore, the architectural effects of these hardware events, which are transparent to user-mode execution, are part of the kernel-mode execution and must be emulated when running the binary analysis for kernel data recovery.

Different hardware events have different architectural effects, and Kernel REPT has to understand the semantics of each hardware event for emulation. We list the information about the hardware events required for software emulation in Table 1. Specifically, Kernel REPT has to not only tell the type of a hardware event, but also infer whether this event occurred in the user or kernel mode and what it was about.

To do so, Kernel REPT first infers the occurrence of a hardware event based on the hardware trace. Given that Kernel REPT only traces the kernel-mode execution, hardware tracing will be paused when the execution returns to the user mode, and resumed when the execution enters the kernel mode. Therefore, the signal of tracing being resumed already indicates the occurrence of a hardware event—a system call or an interrupt/exception happening in the user mode. The hardware trace logs the occurrence of an asynchronous event during the kernel execution. Kernel REPT uses such information to detect the occurrence of an interrupt or exception in the kernel mode.

Next, Kernel REPT needs to infer additional information about a hardware event such as its type and the vector number for an interrupt. The Windows kernel configures the handlers for these hardware events at the boot time and never reconfigures the settings throughout the rest of its lifetime. Kernel REPT assumes this invariant holds for the vast majority of kernel failures under non-adversarial scenarios, and determines the event type as well as the vector number for interrupts/exceptions by comparing the control flow to the kernel configuration stored in the memory dump.

Finally, Kernel REPT emulates the architectural effect of these events according to the hardware specification. Kernel REPT performs the emulation during the binary analysis as if it were emulating a special instruction. However, not all data values are available to Kernel REPT when emulating a hardware event. For example, when emulating an exception from the user mode, Kernel REPT does not know the user-mode instruction that triggers the exception, so it cannot fill up all fields of the trap frame. Similarly, Kernel REPT does not log the parameters of system calls, so it does not necessarily have the register context of a system call event if it cannot be recovered from the memory dump. In these cases, Kernel REPT simply marks the register and memory values as unknown to avoid propagating stale values during the binary analysis.

4 Automatic Analyses

In this section, we present two automatic analyses enabled by Kernel REPT. The first analysis is an automatic root-cause analysis that can identify the buggy function for a specific class of kernel failures. The second analysis is a hybrid analysis that can *proactively* detect bugs that may lead to this class of kernel failures.

4.1 Root-Cause Analysis

A common kernel bug is that calls to *do* operations (e.g., resource acquisition) are *not* matched by calls to *undo* operations (e.g., resource release). For example, if the kernel disables interrupts before entering a critical region but fails to re-enable interrupts after leaving the critical region, the system will crash eventually. Despite the simple nature of this failure type, it is difficult to debug kernel failures caused by these bugs simply based on a memory dump. The key challenge is that the buggy function that missed the *undo* operation may have returned a long time ago. Without an execution history, it is hard to infer which function could be the buggy one. Particularly, the Windows kernel checks if there is a missing *undo* operation (e.g., resource not released) before it returns to the user mode. A failed check leaves developers an empty call stack, which makes it almost impossible to debug. What makes the matter worse is that some operations allow recursion by maintaining a counter for all pending *do* operations (e.g., recursive lock). This requires the developers to match the *do* and *undo* operations in a potentially long history before they can identify the unmatched *do* operation, which further complicates the diagnosis process.

The root-cause analysis identifies the buggy function that misses the *undo* operation by searching along the execution history to find the *first* function where a specified value changes between the function entry and return. For example, to detect when the kernel fails to re-enable the interrupts upon exiting a critical section, the analysis checks for the interrupt enablement at each function entry and return, and reports the first one that has a mismatched value. However, there are exceptions to the above analysis because some functions are designed to just perform the *do* or *undo* operation. For example, if enabling/disabling interrupts is implemented in a library function, then the function is expected to modify the value between its entry and return. The library functions that are designed to perform only a *do* or *undo* operation are relatively stable across kernel versions, so we maintain a whitelist for such functions. The root-cause analysis ignores them when searching for the buggy function.

4.2 Proactive Bug Detection

A common bug pattern that causes *undo* operations being missed is related to the `try/catch`-like primitives designed to handle hardware exceptions gracefully. For example, the Windows kernel uses Structured Exception Handling (SEH) [9] to handle page faults when accessing a user-mode page. An *undo* operation may be missed when an exception occurs if the `try` scope contains a *do* operation and the `catch` scope does not have the corresponding *undo* operation. We show an example of this bug pattern in Figure 3. `foo` calls `read_user_obj` in a `try` block to handle page faults in case the user-mode page is not mapped with proper permissions (line 12). `read_user_obj` temporarily disables Supervisor-Mode Access Prevention (SMAP) in order to access user-mode pages (line 4). If a page fault occurs when `read_user_obj` dereferences `user_ptr`, the page fault handler will redirect the execution back to the `catch` block in `foo` (line 15), skipping the subsequent call to `enable_smap` (line 5). The correct implementation is to apply the scope of

```
1  obj_t read_user_obj(int *user_ptr) {
2      obj_t obj;
3      disable_smap();
4      obj.a = *user_ptr;
5      enable_smap();
6      return obj;
7  }
8
9  int foo() {
10     obj_t obj;
11     try {
12         obj = read_user_obj(user_ptr);
13     }
14     catch {
15         return -1;
16     }
17     return 0;
18 }
```

Figure 3: Example code that misuses `try/catch` leading to a missing *undo* operation.

the `try` block to the dereference of `user_ptr` instead of the entire `read_user_obj` function.

Leveraging the execution history reconstructed by Kernel REPT, we design an automatic *hybrid* analysis to proactively detect bugs of this pattern. Our hybrid analysis has two steps. First, it uses a dynamic analysis to check if there is any *do* operation in a `try` scope based on the execution history. Second, it uses a static analysis to check if the matching `catch` scope does *not* have the corresponding *undo* operation. The analysis identifies `try` and `catch` scopes based on the unwind information in the binaries [14].

The assumption behind this analysis is that a hardware exception may happen at any time within the `try` scope, and missing the *undo* operation in the `catch` scope means that the kernel would fail to restore the state if an exception happens after the *do* operation. Even though this assumption may not be true for all cases, a violation implies an overuse of the `try` scope that should be addressed by developers.

The hybrid analysis is accurate and effective because it leverages execution traces from a huge number of deployed systems. First, a `try` scope may include a significant amount of execution involving multiple levels of function calls. Statically analyzing such a `try` scope is challenging, and our dynamic analysis avoids this challenge. Second, the executions of all kernel threads (i.e., not limited to the failure thread) in a failure report are used in the hybrid analysis. This allows the analysis to avoid the common constraint on completeness for dynamic analysis. Third, the code logic in the `catch` scope is usually straightforward, so simple static analysis is sufficient for checking if an expected *undo* operation exists.

5 Implementation

In this section, we describe the implementation and deployment of Kernel REPT on Microsoft Windows.

5.1 Kernel Tracing

Kernel REPT logs both the context switch history and the control flow of the Windows kernel. To log the context switch history, Kernel REPT leverages Event Tracing for Windows (ETW) [4]. ETW logs the timestamp, identifiers of both the old thread and the new thread for each context switch event. These ETW events will be included in the memory dump of a kernel failure.

To record the kernel's control flow, Kernel REPT enables Intel Processor Trace (PT) [18] on each processor core for the kernel-mode execution at system start. We adapt the driver from REPT to enable Intel PT for the Windows kernel. Our driver change has roughly 2K lines of C code. We mark the virtual memory of trace buffers as read-only to prevent the Windows kernel from accidentally corrupting them. This can be done because Intel PT outputs the trace directly to the physical memory and is not subject to the page permission we set on the virtual memory. Similar to the user-mode tracing in REPT, the kernel-mode trace is stored in the memory dump when a kernel failure is reported.

Kernel REPT currently disables multithreaded analysis for kernel failures due to a caveat of Intel PT. The timestamp logging of Intel PT cannot be configured for a specific privilege level. Without such a privilege-level filtering, the timestamps generated during the user-mode execution will overwrite the kernel's control-flow trace in the circular buffer. One possible solution is to dynamically enable and disable timestamps in software when the processor switches between the user and kernel mode. We leave its exploration to future work.

5.2 Trace Parsing

Intel PT encodes the control flow in a highly compact format. It requires the code binary to parse the trace to reconstruct the control flow. Meanwhile, an operating system can swap out kernel code pages to reclaim its underlying physical memory. This can fail the trace parsing because even capturing the entire physical memory upon a kernel failure can be insufficient due to the unavailability of swapped-out code pages. One possible solution to this problem is to lock all the kernel code pages into the physical memory, but this will increase the memory pressure to the overall system.

In Kernel REPT, we choose to reconstruct the code pages based on the image's metadata stored in the memory dump and its binary file. One disadvantage of this approach is that it does not work for third-party drivers where the binary files are unavailable. This is not a big issue in practice because the code that was executed close to the failure point is usually available in the physical memory (thanks to the memory manager's policy).

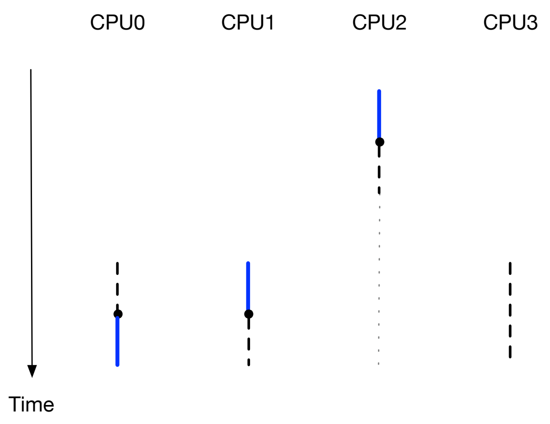

Figure 4: An example scenario where traces of a thread are not contiguous. Solid lines represent the execution trace of the interesting thread. Dashed lines represent the execution trace of other threads. Dots (•) connecting them represent context switches. Dotted lines mean the processor was in sleep mode and no execution trace was generated.

5.3 Binary Analysis

We implement Kernel REPT's binary analysis in 15K lines of C++ code on top of REPT. It includes the emulation of hardware events, the thread trace reconstruction based on the context switch history, and the two automated analyses. Most of the implementation is straightforward, but there are two technical details worth mentioning here.

First, a thread's trace may be *noncontiguous* with respect to the thread's execution. We show an example in Figure 4. In this example, core 2 was in sleep mode for a long period of time, and no execution trace was generated. The trace of the target thread in its circular buffer can be obsolete and disconnected from the rest of the thread's trace on other cores. This can happen when the trace of the target thread on core 1 was overwritten by another thread, and the overwritten trace was more recent than the trace in core 2's trace buffer. Kernel REPT checks for such cases based on the timestamps of context switch events, and discards those disconnected traces.

Second, certain kernel instructions can be missing from the control-flow trace when the system is running inside a virtual machine. In a virtualized environment, guest instructions that can modify the system state (e.g., `wrmsr`) are trapped and emulated by the hypervisor. Kernel REPT only traces the kernel-mode execution, so its binary analysis will be under the illusion that these instructions are skipped according to the Intel PT trace. This can result in an inconsistent state in the data flow recovery. Kernel REPT solves this issue by detecting VMEXIT and adding the skipped instruction back to the instruction sequence if it is deemed as emulated by the hypervisor. Specifically, for each asynchronous event logged in the trace, Kernel REPT checks if there is one and only one possible instruction between the current instruction and the next instruction in the instruction sequence. If so, Kernel REPT determines that the instruction is emulated by the hypervisor, and adds it back to the instruction sequence before running the binary analysis. This check is straightforward to implement because the hypervisor-emulated instruction is typically a non-branch instruction.

5.4 Deployment

We have deployed Kernel REPT in the ecosystem of Microsoft Windows. The deployment of Kernel REPT spans three parts: Windows, Windows Error Reporting (WER) [24], and Windows Debugger [12].

On Windows, we released the kernel driver that configures Intel PT tracing and the ETW context switch logger for the kernel, and a user-mode daemon that communicates with WER to decide when to start/stop the driver. These components were released as part of Windows 10 version 1803.

On the WER service, we added support for requesting Intel PT traces for a given kernel failure. When the WER service receives such a request, it selects devices that have reported the same kernel failures in the past and are capable of Intel PT to enable tracing for future failure reporting. The WER service also runs the automatic root-cause analysis on kernel failures of the specific error code [2].

On Windows Debugger, we implemented Kernel REPT's interactive reverse debugging by extending REPT's debugger extension. This extension allows an developer to set breakpoints, go back and forth on the reconstructed execution history, switch to different threads, and inspect the local and global variables.

6 Evaluation

In this section, we evaluate the performance and effectiveness of Kernel REPT. For performance, we run both micro-benchmarks and real-world applications with Kernel REPT enabled to measure the runtime overhead. For effectiveness, we evaluate Kernel REPT's data recovery and report how it helps developers debug real-world kernel bugs.

6.1 Performance

We evaluate the performance impact of Intel PT tracing and context switch logging on kernel-mode execution by running UnixBench 5.1.3 [11], ApacheBench [1] on Nginx 1.17.5 [7], and JetStream 2 benchmarks [6] on Chrome 79 [3]. We choose UnixBench because it measures the micro-level performance impact on a kernel's key functions

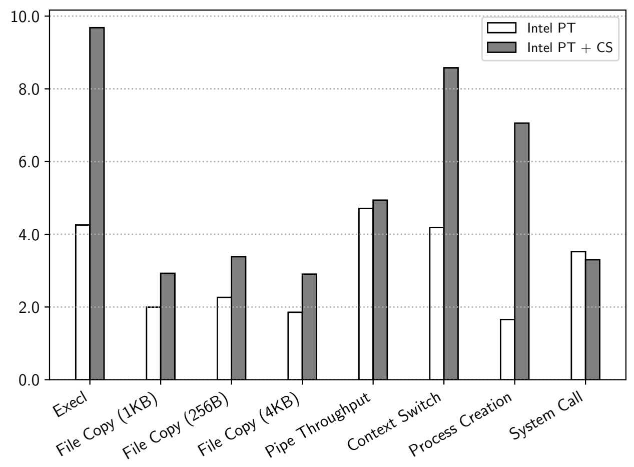

Figure 5: Performance overhead of running UnixBench with Intel PT tracing and context switch logging.

such as system calls and context switches. We choose Nginx and Chrome because they represent popular programs for server and client scenarios, respectively. We run the experiments on a Windows 10 (version 1903) machine equipped with an Intel i7-6700K processor (8 logical cores) and 16GB RAM. We allocate two separate circular buffers for each logical core: a 1MB buffer for Intel PT tracing and a 128KB buffer for context switch logging. We choose this default setting empirically for experiments, but our real-world deployment allows developers to adjust the configuration as needed.

6.1.1 UnixBench

We run UnixBench on the Windows Subsystem for Linux 1 (WSL 1) [13]. WSL 1 implements Linux system calls from the Windows kernel to run unmodified Linux ELF binaries such as UnixBench. We show the performance results of UnixBench in Figure 5. For Intel PT tracing only, the average performance overhead is 3.06% with no single benchmark exceeding 5% overhead. With the context switch logging enabled in addition, the average performance overhead becomes 5.35% with the highest performance overhead of 9.68%. In particular, three benchmarks, Execl, Context Switch and Process Creation, have more frequent context switches than other benchmarks. Therefore, they have higher overhead when context switch logging is turned on.

6.1.2 Nginx

We evaluate the performance overhead of different tracing setups on Nginx using ApacheBench. We run Nginx on the test machine with 8 logical cores and 16GB RAM. We use the default configuration provided by Nginx but modify `worker_processes` to 8 to use all the logical cores. We

	# Instructions	Data Recovery
IRQL Fault (Kernel)	3,310,906	65.13%
Code Overwrite	1,151,315	73.18%
Stack Trash	315,046	65.75%
IRQL Fault (User)	9,176,219	61.56%
Stack Overflow	10,421,430	60.97%
Hardcoded Breakpoint	9,129,048	61.65%
Double Free	5,232,343	43.03%

Table 2: Kernel REPT's data recovery rate on kernel failures caused by `notmyfault` [8].

run ApacheBench on a separate client machine with 16 logical cores. We use the client to make 100,000 HTTP requests over 16 concurrent connections and then measure the throughput (requests/second). We run each session 10 times and report the average throughput. The reduction of the average throughout when the Intel PT tracing is enabled (with or without the context switch logging) is about 2%.

6.1.3 Chrome

We run JetStream 2 benchmarks on the Chrome browser to evaluate the performance impact on web browsing, one of the most common client-side scenarios. There is no visible performance slowdown when both the Intel PT tracing and the context switch logging are enabled. We believe this is because the benchmarks have most of their computation in the user mode.

6.2 Effectiveness

We evaluate the effectiveness of Kernel REPT from four perspectives: (1) how well it can recover data; (2) how its interactive reverse debugging can help developers debug kernel bugs; (3) how accurate the automatic root-cause analysis is; (4) how well the proactive bug detection works.

6.2.1 Data Recovery

We evaluate Kernel REPT's data recovery based on the same metric as REPT [19]. Specifically, we measure the data recovery rate as the percentage of *register uses* (i.e., a register used as the source operand or in the address of a memory operand) for which the register value is recovered by Kernel REPT. Register use is a good metric because it avoids double counting (e.g., we only count it once when `rax` and `rbx` are both known in an instruction `mov rax,rbx`) and memory values are often loaded into registers first before being used in an operation.

In our experiment, we trigger Windows kernel failures by using a public utility program called `notmyfault` [8], which injects a kernel driver to cause various types of failures such as stack overflow and double free. We pick `notmyfault` because its injected failures are reproducible. Our experiment does not include the Buffer Overflow fault in `notmyfault` because it cannot trigger a kernel failure on the latest Windows. For each failure, we count the number instructions and measure the data recovery rate for the crashing thread. Our experimental results are shown in Table 2.

Kernel REPT's data recovery rate is over 60% for all but one failure even when some execution contains over 10 million instructions. The Double Free case involves a series of memory allocation/free operations. As reported in REPT, memory allocation operations are hard to reverse because their metadata may be completely overwritten by subsequent free and reallocation operations. We believe this is the main reason for the Double Free case to have a lower data recovery rate than others.

Comparing with REPT's data recovery on user-mode programs [19, Figure 4], we can see that Kernel REPT achieves a similar data recovery rate for kernel failures. Note that some recovered data may be incorrect, but we cannot directly measure it due to the lack of a ground truth. However, we expect it to be in the same low percentage as REPT.

6.2.2 Interactive Reverse Debugging

We use a real-world case to demonstrate how Kernel REPT can help developers debug kernel bugs through interactive reverse debugging. This bug was introduced to the Windows kernel more than a decade ago. It was not fixed until Kernel REPT became available due to the lack of information in

```
bool get_desc(..., desc_t **p) {
    int size;
    bool success;
    *p = malloc(sizeof(desc_t));
    driver = find_driver();
    success = (driver->op)(*p, &size);
    return success;
}

void foo() {
    desc_t *p;
    bool success;
    if (...) {
        success = get_desc(..., &p));
    } else {
        success = get_desc(..., &p));
    }
    if (success) {
        bar(p->owner->sid); // CRASH!
    }
}
```

Figure 6: A real-world example that showcases how Kernel REPT helps developers find bugs.

memory dumps. We show a simplified version of the code around the bug in Figure 6. In the code snippet, foo calls get_desc to receive a pointer to a descriptor object. Depending on certain conditions (line 13), the call can happen at two places with potentially different parameters (line 14 and 16). get_desc allocates the memory for the descriptor object (line 4) and finds the driver that provides the corresponding callback (line 5). Then, get_desc invokes the driver's callback function to initialize the object, which returns whether the initialization succeeds and the number of bytes being initialized (line 6). Finally, the crash happens when foo dereferences a pointer field (owner) inside the descriptor object (line 19).

To debug this kernel failure, a developer first has to determine where get_desc is called (line 14 or 16). Without Kernel REPT, a developer would need to use some auxiliary information to figure it out. However, with the recorded control flow, it is straightforward to find it. The next step is to determine the target function of the callback (line 6). This can be challenging without the recorded control flow because get_desc has already returned and the relevant information may no longer be available. In fact, the actual code involves multiple levels of indirect function calls, making the problem even harder. With Kernel REPT, the developer can easily reach the correct target function based on the recorded control flow. Finally, the developer has to understand how the descriptor is mis-initialized by the callback function. In this particular case, it turns out that the callback function does not attempt to initialize the descriptor object at all. It just returns success with the number of initialized bytes being zero. Unfortunately, foo does not check the number of bytes being initialized, leading to the subsequent crash caused by dereferencing an uninitialized pointer value. To fix this bug, the developer changes the callback function to return an error code indicating that the operation is not supported.

6.2.3 Root-Cause Analysis

We run the automatic root-cause analysis described in §4.1 on 377 real-world kernel failures of a specific error code [2] reported to Microsoft over two weeks. This error code is used by the kernel when it detects a specific resource is not properly released upon returning to the user mode. The analysis identifies potential buggy functions in 33 kernel components including the core OS kernel, the GUI subsystem, the file system, and some third-party drivers. To evaluate the accuracy of the root-cause analysis, we manually check each identified buggy function either based on the source code or the confirmation from developers. Since the source code of third-party drivers is unavailable and it is difficult to reach their developers, we exclude the 189 kernel failures whose buggy functions are in a third-party driver.

We show the accuracy of the root-cause analysis for the remaining 188 kernel failures in Table 3. The root-cause analysis correctly identifies the buggy function for 148 failures. 136 of these failures are caused by unsafe try/catch operations and 12 of them are caused by other miscellaneous issues (e.g., the code fails to properly clean up the state on an error handling path). The root-cause analysis fails to identify the true buggy function for 40 kernel failures. We manually analyze them via interactive reverse debugging and find that we can find the true buggy function for 23 failures. The rest 17 failures cannot be resolved due to the limited trace size or data recovery.

True Blame		False Blame	
Try/Catch	Misc.	Manual	Unresolved
136	12	23	17

Table 3: The accuracy of the automatic root-cause analysis on 188 real-world kernel failures for which Kernel REPT blames a function in the first-party components.

While analyzing the memory dumps of the 23 failures manually, we find that one common reason for the automatic root-cause analysis to miss the true buggy function is that the *do* and *undo* operation pair is tied to an object's lifetime instead of a function's lifetime. For example, one way to manage the acquisition and release of a lock is to implement the acquire operation in a constructor function and the release operation in the corresponding destructor function. In this case, a function can indirectly acquire the lock by creating such an object, and then passes it to another function that releases this lock by destructing the object. If the kernel fails to destruct the object due to programming errors, it not only causes memory leaks but also leads to the missing *undo* issue. The root cause of such programming errors can vary case by case, and blaming the function that creates the object and seemingly misses the destruction does not always lead to the correct outcome. However, even in these cases, the root-cause analysis can provide useful information to help developers find the root cause.

6.2.4 Proactive Bug Detection

We run the proactive bug detection described in §4.2 over 2000 execution histories reconstructed from memory dumps of real-world kernel failures. We do not limit the failure type, and use the execution histories of *all* threads in a memory dump. We use memory dumps of kernel failures instead of normal executions because they are currently the major source of recorded real-world kernel executions.

We have found a total of 17 previously unknown kernel bugs, and all are confirmed and fixed. For one of the bugs, we observed an actual kernel failure caused by it a few days after we reported it to the developer. This shows the potential of using Kernel REPT to uncover bugs even before they are triggered in practice.

7 Discussion

Kernel REPT extends REPT's support for reverse debugging of user-mode failures to kernel-mode failures. Therefore, it shares two limitations with REPT. First, the reconstructed execution history is incomplete because the circular trace buffer only captures a fixed amount of control-flow information before the kernel failure. Second, the reconstructed execution history is imperfect because many instructions are not reversible. Despite the two limitations, Kernel REPT's reverse debugging capability allows successful diagnosis of many real-world kernel failures that were impossible to debug before.

The automatic root-cause analysis described in §4.1 requires a whitelist of functions that perform only a *do* or *undo* operation by design. It requires manual effort to construct and update the whitelist. The root-cause analysis may have false blames due to the incompleteness of the whitelist. In practice, we rely on developers' feedback to keep the whitelist up to date.

One interesting observation we have is that REPT-style reverse debugging is more effective for kernel-mode failures than for user-mode failures. We believe the key reason is that the Windows kernel operates in a more *defensive* manner: it performs various checks of invariants in kernel state at different times of the execution, such as checking missing undo operations before returning to the user mode (see §4.1 for details). These checks shorten the execution between the program defect and the program failure. This is crucial for the effectiveness of REPT-style reverse debugging since its reconstructed execution history is incomplete and imperfect.

8 Related Work

In this section, we discuss the previous work related to Kernel REPT in three categories: record and replay, failure analysis and failure reproduction. We omit the discussion of REPT [19] as we have covered it comprehensively in §2.1.

8.1 Record and Replay

Record and replay tools have been applied to debugging for both user-mode applications [15, 25, 30, 33] and operating systems [16,21–23,28,31,32,34,35]. Software-based record and replay tools [16, 21–23, 28, 35] for operating systems require running the whole system in a virtualized environment to log all non-deterministic inputs to the target system. These tools are rarely deployed in production environments because of their significant runtime overhead and compatibility issue. The latter is caused by the requirement for a special setup such as installing a custom virtual machine.

Hardware-based record and replay tools [31, 32, 34] modify the underlying hardware to record the execution of a target system. For example, Flight Data Recorder (FDR) [34] instruments the processor's cache coherency protocol to enable record and replay of a multiprocessor system. The required hardware modification makes these systems expensive to build and adopt in practice.

Compared to the above record and replay tools, Kernel REPT enables effective reverse debugging for operating systems running on commodity hardware with low performance and space overhead at runtime, making it practical for deployment on real-world systems.

8.2 Failure Analysis

RETracer [20] is a triaging system for both user-mode and kernel-mode failures. It starts with a corrupted pointer, performs backward taint analysis on memory dumps, and assigns the blame to a function that contributes to the access violation. RETracer uses the crashing stack as an approximate execution trace when performing backward taint analysis, so it cannot effectively analyze kernel failures with an empty call stack.

Postmortem Symbolic Evaluation (PSE) [29] performs static backward slicing on memory dumps to identify where a bad pointer is originated. PSE is also limited by the information available in the dump, and can have false positives due to unresolved memory aliases.

SherLog [36] analyzes the log messages generated during a failed execution to infer control and data values before the failure point. While this approach may be useful to diagnose logical bugs in a program, log messages cannot be used to diagnose low-level software bugs such as memory safety errors. In addition, its effectiveness depends on the developer's expertise in determining the key information to log, which varies case by case.

Kernel REPT is complementary to the above production failure analysis techniques. For instance, we integrated REPT-style reverse debugging into RETracer so that the latter can run its backward taint analysis on the reconstructed execution history to derive a more precise blame for production failures.

8.3 Failure Reproduction

Execution Synthesis (ESD) [37] explores possible program paths to search for inputs that can lead to the same failure. ESD relies solely on memory dumps, and its symbolic execution [17] may not be able to solve complicated constraints when exploring a long execution history of complex program state. This makes it difficult to work for complex programs such as the operating system kernel.

BugRedux [26] reproduces a production failure by instrumenting the program to collect execution data at different levels and employing symbolic execution to compute an input leading to a similar execution. Program instrumentation incurs overhead even for normal executions, and symbolic execution is known to have path explosion problems.

Kernel REPT allows developers to examine the execution history of a kernel failure without the need to reproduce it.

9 Conclusion

We have presented the design and implementation of Kernel REPT, the first practical solution for reverse debugging of kernel failures in deployed systems. Kernel REPT records the kernel's control flow and context switch events on each processor, and recovers its data flow on each software thread via binary analysis. Its analysis emulates both machine instructions and hardware events such as interrupts and exceptions. In addition to the support for interactive reverse debugging, we have developed two automatic analyses on top of Kernel REPT: a root-cause analysis that can identify the buggy function for a class of kernel failures, and a hybrid analysis that can proactively detect bugs due to a misuse of the `try/catch` primitive. We show that Kernel REPT is efficient for real-world deployment and effective for debugging real-world kernel failures.

Acknowledgments

We thank our shepherd, David Devecsery, and other reviewers for their insightful feedback during this unprecedented time. We are also very grateful for all the help from our colleagues at Microsoft during the development and deployment of Kernel REPT. In particular, we thank Alan Auerbach, Joe Ballantyne, Harini Bandi, Abolade Gbadegesin, Stephen Hufnagel, Mehmet Iyigun, Pedro Justo, Scott Lee, Graham McIntyre, Timothy Misiak, Viorel Mititean, Jordi Mola and Pin Wu for their tremendous help in making this project a fully deployed end-to-end system.

References

[1] ApacheBench: A Complete Benchmarking and Regression Testing Suite. https://httpd.apache.org/docs/2.2/programs/ab.html.

[2] APC Index Mismatch. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/bug-check-0x1--apc-index-mismatch.

[3] Chrome. https://www.google.com/chrome/.

[4] Event Tracing for Windows (ETW). https://docs.microsoft.com/en-us/windows/win32/etw/about-event-tracing.

[5] GDB and Reverse Debugging. https://www.gnu.org/software/gdb/news/reversible.html.

[6] JetStream 2. https://browserbench.org/JetStream/.

[7] Nginx. https://www.nginx.com/.

[8] NotMyFault. https://docs.microsoft.com/en-us/sysinternals/downloads/notmyfault.

[9] Structured Exception Handling (SEH). https://docs.microsoft.com/en-us/cpp/cpp/structured-exception-handling-c-cpp.

[10] UndoDB: The Interactive Reverse Debugger for C/C++ on Linux and Android. https://undo.io/.

[11] UnixBench. https://github.com/kdlucas/byte-unixbench.

[12] Windows Debugger. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/.

[13] Windows Subsystem for Linux (WSL). https://docs.microsoft.com/en-us/windows/wsl/about.

[14] x64 Exception Handling. https://docs.microsoft.com/en-us/cpp/build/exception-handling-x64.

[15] Gautam Altekar and Ion Stoica. ODR: Output-Deterministic Replay for Multicore Debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 193–206. ACM, 2009.

[16] Prashanth P Bungale and Chi-Keung Luk. PinOS: A Programmable Framework for Whole-System Dynamic Instrumentation. In *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE)*, pages 137–147. ACM, 2007.

[17] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[18] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual.

[19] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. REPT: Reverse Debugging of Failures in Deployed Software. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–32, 2018.

[20] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P. Kemerlis. RETracer: Triaging Crashes by Reverse Execution from Partial Memory Dumps. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 2016.

[21] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable Reverse Engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*. ACM, 2015.

[22] George W Dunlap, Samuel T King, Sukru Cinar, Murtaza A Basrai, and Peter M Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[23] George W Dunlap, Dominic G Lucchetti, Michael A Fetterman, and Peter M Chen. Execution Replay of Multiprocessor Virtual Machines. In *Proceedings of the 4th International Conference on Virtual Execution Environments (VEE)*. ACM, 2008.

[24] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[25] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M Frans Kaashoek, and Zheng Zhang. R2: An Application-Level Kernel for Record and Replay. In *Proceedings of the 8th USENIX SYmposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[26] Wei Jin and Alessandro Orso. BugRedux: Reproducing Field Failures for In-House Debugging. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, 2012.

[27] Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. Lazy Diagnosis of In-Production Concurrency Bugs. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[28] Samuel T King, George W Dunlap, and Peter M Chen. Debugging Operating Systems with Time-Traveling Virtual Machines. In *Proceedings of the 2005 USENIX Annual Technical Conference (ATC)*, 2005.

[29] Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. PSE: Explaining Program Failures via Postmortem Static Analysis. In *Proceedings of the 12th ACM SIGSOFT 12th International Symposium on Foundations of Software Engineering (FSE)*, 2004.

[30] Ali Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. Towards Practical Default-On Multi-Core Record/Replay. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[31] Pablo Montesinos, Luis Ceze, and Josep Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA)*, 2008.

[32] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, 2005.

[33] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering Record and Replay for Deployability. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, 2017.

[34] Min Xu, Rastislav Bodik, and Mark D Hill. A Flight Data Recorder for Enabling Full-System Multiprocessor Deterministic Replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*. ACM, 2002.

[35] Min Xu, Vyacheslav Malyugin, Jeffrey Sheldon, Venkitachalam Weissman, Ganesh, and Boris Weissman. Retrace: Collecting Execution Trace with Virtual Machine Deterministic Replay. In *Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, 2007.

[36] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. SherLog: Error Diagnosis by Connecting Clues from Run-Time Logs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[37] Cristian Zamfir and George Candea. Execution Synthesis: A Technique for Automated Debugging. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*, 2010.