Recovering from Faulty Device Drivers

Nathanael Paul **David Evans**

Department of Computer Science
University of Virginia
Charlottesville, VA
{nate, evans}@cs.virginia.edu

With current operating systems, recovering from a device driver failure usually requires rebooting. The goal of our work is to develop mechanisms operating systems can use to automatically detect and recover from errors in device drivers.

Several studies (see Swift et. al.'s study of Windows XP in SOSP 2003 and Chou et. al's study of Linux in SOSP 2001) have attributed a large fraction of operating system failures to device driver flaws. Not only can driver errors cause kernel instability, but these errors can also be exploited for privilege escalation and access to kernel data structures. A search on securityfocus.com shows vulnerabilities and advisories regarding device drivers including dozens of types of errors: buffer overflows, integer overflows, denial of service, and other vulnerabilities.

Many of these failures stem from driver code failing to follow API requirements. Several projects have developed techniques for detecting violations of these requirements using static analysis, most notably the SLAM project which Microsoft has incorporated into the DDK as the Static Driver Verifier (SDV). Although static approaches can be very useful, there are limitations on what can be done with purely static analysis. Statically analyzing source code involves checking specific properties through different executable code paths, and there are often false positives and limitations on the number of properties that can be checked while balancing the possibly exponential number of code paths. With SDV, a table is produced categorizing the errors found in its analysis, and the developer must then fix those errors. These static approaches require cooperation from the device driver developer, and there are no guarantees the developer will fix an error if the driver appears to work or even run the static tools. Hence, we propose a tool that dynamically checks an executing driver and dynamically recovers in case of driver error. Our approach is complementary to static approaches (when they are used), but provides end users with additional control over driver properties and stronger guarantees.

In the past recovery has been primarily done in two ways: checkpointing or a reset of total system state. Many users reboot to recover from an erroneous state, and some applications use checkpointing, reverting to what is thought to be an error-free state, for recovery. Demsky and Rinard (OOPSLA 2003) developed a recovery technique for detecting violated invariants within faulty data structures, and then repairing the data structures by dynamically fixing them to meet the specified invariant(s). Our design will not monitor low-level data structure invariants, but instead use the driver's current and past use of kernel APIs to recover from an error. For example, one API constraint is that a spinlock should be released before blocking. To recover from a call to block with a currently held spinlock, we catch the premature call to block and inject a call to safely release the spinlock before blocking.

We have started an implementation using .NET's Rotor source code. For API monitoring, we are injecting code into sample programs just before a monitored method is compiled from its intermediate form allowing our solution to be easily ported. Our future work includes integrating the kernel's API rules, trying different methods of recovery, forming different policies, and experimenting with different designs to improve performance.

Since the dynamic checking and recovering is intended for production environments, performance is critical. End users browsing the web and checking email will have different requirements than those using their computer for a development machine. We hope to produce a system that can take existing high performance video and network device drivers and execute them safely and efficiently. By providing efficient mechanisms for recovery of device drivers within the kernel, there exists potential to greatly improve the reliability and security of mainstream operating systems.