Testing Closed-Source Binary Device Drivers with DDT

Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract

DDT is a system for testing closed-source binary device drivers against undesired behaviors, like race conditions, memory errors, resource leaks, etc. One can metaphorically think of it as a pesticide against device driver bugs. DDT combines virtualization with a specialized form of symbolic execution to thoroughly exercise tested drivers; a set of modular dynamic checkers identify bug conditions and produce detailed, executable traces for every path that leads to a failure. These traces can be used to easily reproduce and understand the bugs, thus both proving their existence and helping debug them. We applied DDT to several closed-source Microsoft-certified Windows device drivers and discovered 14 serious new bugs. DDT is easy to use, as it requires no access to source code and no assistance from users. We therefore envision DDT being useful not only to developers and testers, but also to consumers who want to avoid running buggy drivers in their OS kernels.

1 Introduction

Device drivers are one of the least reliable parts of an OS kernel. Drivers and other extensions—which comprise, for instance, 70% of the Linux operating system—have a reported error rate that is 3-7 times higher than the rest of the kernel code [11], making them substantially more failure-prone. Not surprisingly, 85% of Windows crashes are caused by driver failures [27]. Moreover, some drivers are vulnerable to malformed input from untrusted user-space applications, allowing an attacker to execute arbitrary code with kernel privilege [5].

It is therefore ironic that most computer users place full trust in closed-source binary device drivers: they run drivers (software that is often outsourced by hardware vendors to offshore programmers) inside the kernel at the highest privilege levels, yet enjoy a false sense of safety by purchasing anti-virus software and personal firewalls. Device driver flaws are more dangerous than application vulnerabilities, because device drivers can subvert the entire system and, by having direct memory access, can be used to overwrite both kernel and application memory. Recently, a zero-day vulnerability within a driver shipped with all versions of Windows allowed non-privileged users to elevate their privileges to Local System, leading to complete system compromise [24].

Our goal is to empower users to thoroughly test drivers before installing and loading them. We wish that the Windows pop-up requesting confirmation to install an uncertified driver also offered a “Test Now” button. By clicking that button, the user would launch a thorough test of the driver’s binary; this could run locally or be automatically shipped to a trusted Internet service to perform the testing on behalf of the user. Such functionality would benefit not only end users, but also the IT staff charged with managing corporate networks, desktops, and servers using proprietary device drivers.

Our work applies to all drivers, including those for which source code is not available, thus complementing the existing body of driver reliability techniques. There exist several tools and techniques that can be used to build more reliable drivers [14, 23, 1] or to protect the kernel from misbehaving drivers [30], but these are primarily aimed at developers who have the driver’s source code. Therefore, these techniques cannot be used (or even adapted) for the use of consumers on closed-source binary drivers. Our goal is to fill this gap.

We believe that the availability of consumer-side testing of device drivers is essential. As of 2004, there were 800,000 different kinds of PnP devices at customer sites, with 1,500 devices being added every day [26]. There were 31,000 unique drivers, and 9 new drivers were released every day. Each driver had ~3.5 versions in the field, with 88 new driver versions being released every day [26]. Faced with an increasing diversity of drivers, consumers (end users and IT specialists alike) need a way to perform end-to-end testing just before installation.

This paper describes DDT, a device driver testing system aimed at addressing these needs. DDT uses selective symbolic execution to explore the device driver’s execution paths, and checks whether these paths can cause undesired behavior, such as crashing the kernel or overflowing a buffer. For each suspected case of bad behavior, DDT produces a replayable trace of the execution that led to the bug, providing the consumer irrefutable evidence of the problem. The trace can be re-executed on its own, or inside a debugger.

DDT currently works for Windows device drivers. We applied it to six popular binary drivers, finding 14 bugs with relatively little effort. These include race conditions, memory bugs, use of unchecked parameters, and resource leaks, all leading to kernel crashes or hangs. Since DDT found bugs in drivers that have successfully passed Microsoft certification, we believe it could be used to improve the driver certification process.

Our work makes two contributions: (1) The first system that can thoroughly and automatically test closed-source binary drivers, without access to the corresponding hardware device; (2) The concept of fully symbolic hardware—including symbolic interrupts—and demonstration of its use for testing kernel-mode binaries.

The rest of the paper is structured as follows: §2 provides a high-level overview of DDT, §3 describes the design in detail, §4 presents our current DDT prototype for Windows drivers, §5 evaluates DDT on six closed-source device drivers, §6 discusses limitations of our current prototype, §7 reviews related work, and §8 concludes.

2 Overview

DDT takes as input a binary device driver and outputs a report of found bugs, along with execution traces for each bug. The input driver is loaded in its native, unmodified environment, which consists of the OS kernel and the rest of the software stack above it. DDT then exercises automatically the driver along as many code paths as possible, and checks for undesired properties. When an error or misbehavior is detected, DDT logs the details of the path exploration along with an executable trace. This can be used for debugging, or merely as evidence to prove the presence of the bug.

DDT has two main components: a set of pluggable bug checkers and a driver exerciser (Figure 1). The exerciser is in charge of steering the driver down various execution paths—just like a personal trainer, it forces the driver to exercise all the various ways in which it can run. The dynamic checkers watch the execution and flag undesired driver behaviors along the executed paths. When they notice a bug, they ask the exerciser to produce information on how to reach that same situation again.

DDT provides a default set of checkers, and this set can be extended with an arbitrary number of other checkers for both safety and liveness properties (see §3.1). Currently, DDT detects the following types of bugs: memory access errors, including buffer overflows; race conditions and deadlocks; incorrectly handled interrupts; accesses to pageable memory when page faults are not allowed; memory leaks and other resource leaks; mishandled I/O requests (e.g., setting various I/O completion flags incorrectly); any action leading to kernel panic; and incorrect uses of kernel APIs.

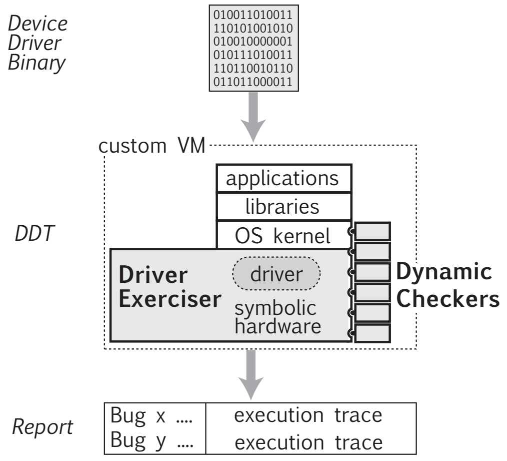

Figure 1: DDT's VM-based architecture.

These default checkers catch the majority of defects in the field. Ganapathi et al. found that the top driver problems causing crashes in Windows were 45% memory-related (e.g., bad pointers), 15% poorly handled exceptions, 13% infinite loops, and 3% unexpected traps [15]. A Microsoft report [26] found that, often, drivers crash the system due to not checking for error conditions following a call to the kernel. It is hypothesized that this is due to programmers copy-pasting code from the device driver development kit's succinct examples.

Black-box testing of closed-source binary device drivers is difficult and typically has low code coverage. This has two main reasons: First, it is hard to exercise the driver through the many layers of the software stack that lie between the driver's interface and the application interface. Second, closed-source programs are notoriously hard to test as a black box. The classic approach to testing such drivers is to try to produce inputs that exercise as many paths as possible and (perhaps) check for high-level properties (e.g., absence of kernel crashes) during those executions. Considering the wide range of possible inputs and system events that are hard to control (e.g., interrupts), this approach exercises relatively few paths, thus offering few opportunities to find bugs.

DDT uses selective symbolic execution [10] of the driver binary to automatically take the driver down as many paths as possible; the checkers verify desired properties along these paths. Symbolic execution [20, 6, 7] consists of providing a program with symbolic inputs (e.g., α or β) instead of concrete ones (e.g., 6 or "abc"), and letting these values propagate as the program executes, while tracking path constraints (e.g., $\beta = \alpha + 5$). When a symbolic value is used to decide the direction of a conditional branch, symbolic execution explores all feasible alternatives. On each branch, a suitable path constraint is added on the symbolic value to ensure its set of possible values satisfies the branch condition (e.g., $\beta < 0$ and $\beta >= 0$, respectively). Selective symbolic execution enables the symbolic execution of one piece of the software stack (the device driver, in our case) while the rest of the software runs concretely.

A key challenge is keeping the symbolic and the concrete portions of the execution synchronized. DDT supplies the driver with symbolic values on the calls from the kernel to the driver (§3.2) as well as on the returns from the hardware to the driver (§3.3), thus enabling an underlying symbolic execution engine to steer the driver on the various possible paths. When the driver returns values to a kernel-originated call, or when the driver calls into the kernel, parameters and driver are converted so that execution remains consistent, despite the alternation of symbolic and concrete execution.

DDT's *fully symbolic hardware* enables testing drivers even when the corresponding hardware device is not available. DDT never calls the actual hardware, but instead replaces all hardware reads with symbolic values, and discards all writes to hardware. Being able to test a driver without access to the hardware is useful, for example, for certification companies that cannot buy all the hardware variants for the drivers they test, or for consumers who would rather defer purchasing the device until they are convinced the driver is trustworthy.

Symbolic hardware also enables DDT to explore paths that are hard to test without simulators or specialized hardware. For example, many devices rely on interrupts to signal completion of operations to the device driver. DDT uses *symbolic interrupts* to inject such events at the various crucial points during the execution of the driver. Symbolic interrupts allow DDT to test different code interleavings and detect bugs like the race conditions described in §5.1.

DDT provides evidence of the bug and the means to debug it: a complete trace of the execution plus concrete inputs and system events that make the driver re-execute the buggy path in a regular, non-DDT environment.

3 Design

We now present DDT's design, starting with the types of bugs DDT looks for (§3.1), an overview of how drivers are exercised (§3.2), a description of fully symbolic hardware (§3.3), the use of annotations to extend DDT's capabilities (§3.4), and finally we show how generated traces are used to replay bugs and fix them (§3.5).

3.1 Detecting Undesired Behaviors

DDT uses two methods to detect failures along exercised paths: dynamic verification done by DDT's virtual machine (§3.1.1) and failure detection inside the guest OS (§3.1.2). VM-level checks are targeted at properties that require either instrumentation of driver code instructions or reasoning about multiple paths at a time. Guest OS-level checks leverage existing stress-testing and verification tools to catch bugs that require deeper knowledge of the kernel APIs. Most guest OS-level checks can be performed at the VM level as well, but it is often more convenient to write and deploy OS-level checkers.

3.1.1 Virtual Machine-Level Checks

Memory access verification in DDT is done at the VM level. On each memory access, DDT checks whether the driver has sufficient permissions to access that memory. For the purpose of access verification, DDT treats the following memory regions as accessible to drivers:

- Dynamically allocated memory and buffers;
- Buffers passed to the driver, such as network packets or strings from the Windows registry;
- Global kernel variables that are implicitly accessible to drivers;
- Current driver stack (accesses to memory locations below the stack pointer are prohibited, because these locations could be overwritten by an interrupt handler that saves context on the stack);
- Executable image area, i.e., loadable sections of the driver binary with corresponding permissions;
- Hardware-related memory areas (memory-mapped registers, DMA memory, or I/O ranges).

In order to track these memory regions, DDT hooks the kernel API functions and driver entry points. Every time the hooked functions are called, DDT analyzes their arguments to determine which memory was granted to (or revoked from) the driver. The required knowledge about specific kernel APIs can be provided through lightweight API annotations (see §3.4).

Beyond memory safety, DDT's simultaneous access to multiple execution paths (by virtue of employing symbolic execution) enables the implementation of bug detection techniques that reason about the code globally in terms of paths, such as infinite loop detection [34].

3.1.2 Guest Operating System-Level Checks

In addition to VM-level checkers, DDT can also reuse off-the-shelf runtime verification tools. These tools perform in-guest checking, oblivious to exactly how the driver is being driven along the observed execution paths. Since these tools are usually written by OS developers (e.g., for driver certification programs, like Microsoft's WHQL [25]), they can detect errors that require deep knowledge of the OS and its driver API.

When they find a bug, these dynamic tools typically crash the system to produce an error report containing a memory dump. DDT intercepts such premeditated crashes and reports the bug information to the user. DDT helps the runtime checkers find more bugs than they would do under normal concrete execution, because it symbolically execute the driver along many more paths.

DDT's modular architecture (Figure 1) allows reusing such tools without adaptation or porting. This means that driver developers' custom test suites can also be readily employed. Moreover, given DDT's design, such tools can be inserted at any level in the software stack, either in the form of device drivers or as software applications.

DDT can also automatically leverage kernel assertion checks, when they are present. For example, the checked build version of Windows contains many consistency checks—with DDT, these assertions get a better chance of being exercised along different paths.

3.2 Exercising the Driver: Kernel/Driver Interface

DDT implements selective symbolic execution [10], a technique for seamless transfer of system state between symbolic and concrete phases of execution. DDT obtaines similar properties to running the entire system symbolically, while in fact only running the driver symbolically. The transfer of state between phases is governed by a set of conversion hints (see §3.4). Using selective symbolic execution enables DDT to execute the driver within its actual environment, as opposed to requiring potentially incomplete models thereof [1, 6].

A typical driver is composed of several entry points. When the OS loads the driver, it calls its main entry point, similarly to a shell invoking the `main()` function of a program. This entry point registers with the kernel the driver's other entry points. For example, a typical driver would register `open`, `read`, `write`, and `close` entry points, which are then called by the OS when a user-mode application makes use of the driver.

When the kernel calls a driver's entry point, DDT transfers system state to a symbolic execution engine. It converts entry point arguments, and possibly other parts of concrete system state, to symbolic values, according to the annotations described in §3.4. For example, when the kernel calls the `SendPacket` function in a NIC driver, DDT makes the content of the network packet symbolic, to explore all the paths that depend on the packet's type.

When a driver calls a kernel function, DDT selects feasible values (at random) for its symbolic arguments. For example, if the driver calls the `AllocatePool` function with a symbolic length argument, DDT selects some concrete value *len* for the length that satisfies current constraints. However, this concretization subjects all subsequent paths to the constraint that length must equal *len*, and this may disable otherwise-feasible paths. Thus, DDT keeps track of all such concretization-related constraints and, if at some point in the future this constraint limits a choice of paths, DDT backtracks to the point of concretization, forks the entire machine state, and repeats the kernel call with different feasible concrete values, which could re-enable the presently unexplorable path.

To minimize overhead, DDT does concretization on-demand, i.e., delays it as long as possible by tracking symbolic values when executing in concrete mode and concretizing them only when they are actually read. This way, symbolic values that are not accessed by concretely running code are never concretized. In particular, all private driver state and buffers that are treated as opaque by the kernel end up being preserved in their symbolic form.

3.3 Exercising the Driver: Symbolic Hardware

DDT requires neither real hardware nor hardware models to test drivers—instead, DDT uses symbolic hardware. A symbolic device in DDT ignores all writes to its registers and produces symbolic values in response to reads. These symbolic values cause drivers to explore paths that depend on the device output.

Symbolic hardware produces symbolic interrupts, i.e., interrupts with a symbolic arrival time. Reasoning about interrupt arrival symbolically offers similar benefits to reasoning about program inputs symbolically: the majority of interrupt arrival times are equivalent to each other, so only one arrival time in each equivalence class need be produced. If a block of code does not read/write system state that is also read/written by the interrupt handler, then executing the interrupt handler at any point during the execution of that block has the same end result.

Currently, DDT implements a simplified model of symbolic interrupts. It symbolically delivers interrupts on each crossing of the kernel/driver boundary (i.e., before and after each kernel API call, and before and after each driver entry point execution). While not complete, we found that this strategy produces good results, because many important changes in driver state are related to crossing the kernel/driver interface.

Symbolic hardware with symbolic interrupts may force the driver on paths that are not possible in reality with correct hardware. For example, a symbolic interrupt may be issued after the driver instructed the device not to issue interrupts (e.g., by writing a control register). A correctly functioning device will therefore not deliver that interrupt. The natural solution would be to include the enabled/disabled interrupts status in the path constraints, and prevent interrupts from occurring when this is not possible. However, recent work [19] has shown that hardware often malfunctions, and that drivers must be sufficiently robust to handle such behavior anyway.

More generally, DDT's ability to test drivers against hardware failures is important, because chipsets often get revised without the drivers being suitably updated. Consider a device that returns a value used by the driver as an array index. If the driver does not check the bounds (a common bug [19]) and a revised version of the chipset later returns a greater value, then the obsolete driver could experience an out-of-bounds error.

3.4 Enabling Rich Driver/Environment Interactions

Device drivers run at the bottom of the software stack, sandwiched between the kernel and hardware devices. The layers surrounding a driver are complex, and the different classes of device drivers use many different kernel subsystems. For instance, network, audio, and graphics drivers each use different kernel services and interfaces.

One may be tempted to run drivers in isolation for purposes of testing. Unfortunately, this requires an abstraction layer between the drivers and the rest of the stack, and building this layer is non-trivial. For example, testing a network driver would require the testbed to provide well-formed data structures when returning from a packet allocation function called by the driver.

DDT tests drivers by symbolically executing them in conjunction with the real kernel binary. By using the actual software stack (and thus the real kernel) instead of a simplified abstract model of it, DDT ensures that the device drivers get tested with the exact kernel behavior they would experience in reality. To this end, DDT needs to mediate the interactions with the layers around the driver in a way that keeps the symbolic execution of the driver consistent with the concrete execution of the kernel.

DDT performs various conversions between the symbolic and concrete domains. In its default mode, in which no annotations are used, DDT converts symbolic arguments passed to kernel functions into legal random concrete values and uses symbolic hardware, including symbolic interrupts. Driver entry point arguments are not touched. These conversions, however, can be fine-tuned by annotating API functions and driver entry points.

3.4.1 Extending DDT with Interface Annotations

DDT provides ways for developers to encode their knowledge of the driver/kernel API in annotations that improve DDT's achievable code coverage and bug finding abilities. Annotations allow DDT to detect not only low-level errors, but also logical bugs. Annotations are a one-time effort on the part of OS developers, testers, or a broader developer community.

The idea of encoding API usage rules in annotations is often used by model checking tools, with a recent notable example being SLAM [1]. However, DDT's annotations are lighter weight and substantially easier to write and keep up-to-date than the API models used by previous tools: preparing DDT annotations for the whole NDIS API, which consists of 277 exported functions, took about two weeks of on-and-off effort; preparing annotations for those 54 functions in the WDM API that were used by our sound drivers took one day.

DDT annotations are written in C and compiled to LLVM bitcode [22], which is then loaded by DDT at runtime and run in the context of QEMU-translated code, when necessary. The annotation code has direct access to, and control over, the guest system's state. Additionally, it can use a special API provided by DDT to create symbolic values and/or manipulate execution state.

The following annotation introduces positive integer symbolic values when the driver reads a configuration parameter from the Windows registry:

```
 1 void NdisReadConfiguration_return(CPU* cpu){
 2   if(*((PNDIS_STATUS) ARG(cpu, 0)) == 0
 3            && ARG(cpu, 4) == 1) {
 4     int symb = ddt_new_symb_int();
 5     if(symb >= 0)
 6       ((PNDIS_CONFIGURATION_PARAMETER)
 7         ARG(cpu, 1))->IntegerData = symb;
 8     else ddt_discard_state();
 9   }
10 }
```

This sample annotation function is invoked on the return path from `NdisReadConfiguration` (hence its name—line 1). It checks whether the call returned successfully (line 2) and whether the type of the value is integer (line 3). It then creates an unconstrained symbolic integer value using DDT's special API (line 4), after which it checks the value (line 5) and discards the path on which *symb* is not a positive integer (line 8).

DDT annotations fall into four categories:

Concrete-to-symbolic conversion hints apply to driver entry points' arguments and to return values from kernel functions called by the driver. They encode contracts about what constitute reasonable arguments or return values. For example, a memory allocation function can either return a valid pointer or a null pointer, so the annotation would instruct DDT to try both the originally returned concrete pointer, as well as the null-pointer alternative. The absence of this kind of conversion hints will cause DDT not to try all reasonable classes of values, which results solely in decreased coverage, i.e., false negatives.

Symbolic-to-concrete conversion hints specify the allowed set of values for arguments to kernel API functions called by drivers. They include various API usage rules that, if violated, may lead to crashes or data corruption. When a call to such an annotated function occurs, DDT verifies that all incorrect argument values are ruled out by the constraints on the current path; if not, it flags a potential bug. The absence of such annotations can lead DDT to concretize arguments into some values that are consistent with the path constraints (thus feasible in a real execution) but not uncover potential bugs (if the values happen to be OK according to the unspecified API usage rules). In other words, they can lead to false negatives, but not to false positives.

Resource allocation hints specify whether invoking an entry point or calling a kernel function grants or revokes the driver's access to any memory or other resources. This information is used to verify that the driver accesses only resources that the kernel explicitly allows it to access. It is also used to verify that all allocated resources are freed on exit paths. The absence of memory allocation hints can lead to false positives, but can be avoided, if necessary, by switching to a coarse-grained memory access verification scheme (as used, for instance, in Microsoft's Driver Verifier [25]).

Kernel crash handler hook: This annotation informs DDT of the address of the guest kernel's crash handler, as well as how to extract the crash information from memory. This annotation enables DDT to intercept all crashes when running the kernel concretely, such as the "blue screen of death" (BSOD) on Windows. This annotation is relied upon in our DDT prototype to cooperate with the Microsoft Driver Verifier's dynamic checkers.

3.4.2 Alternative Approaches

We have gone to great lengths to run the drivers in a real environment and avoid abstract modeling. Is it worth it?

One classic approach to ensuring device driver quality is stress-testing, which is how Microsoft certifies its third-party drivers [25]. However, this does not catch all bugs. As we shall see in the evaluation, even Microsoft-certified drivers shipped with Windows have bugs that cause the kernel to crash. However, powerful static analysis tools [1] can reason about corner-case conditions by abstracting the driver under test, without actually running it. Since static analysis does not run any code per se, it requires modeling the driver's environment.

We believe environment modeling generally does not scale, because kernels are large and evolve constantly. Modeling the kernel/driver API requires manual effort and is error prone. According to [1], developing around 60 API usage rules for testing Windows device drivers took more than three years. It also required many iterations of refinement based on false positives found during evaluation. In the end, the resulting models are only an approximation of the original kernel code, thus leading to both false negatives and, more importantly, false positives. A test tool that produces frequent false positives discourages developers from using it.

In contrast, we find DDT's annotations to be straightforward and easy to maintain. Moreover, if they are perceived by developers as too high of a burden, then DDT can be used in its default mode, without annotations.

Testing device drivers often requires access to either the physical device or a detailed model of it. For drivers that support several physical devices, testing must be repeated for each such device. In contrast, symbolic hardware enables not only testing drivers without a physical device, but also testing them against hardware bugs or corner cases that are hard to produce with a real device.

3.5 Verifying and Replaying Bugs

When DDT finishes testing a driver, it produces a detailed report containing all the bugs it found. This report consists of all faulty execution paths and contains enough information to accurately replay the execution, allowing the bug to be reproduced on the developer's or consumer's machine.

DDT's bug report is a collection of traces of the execution paths leading to the bugs. These traces contain the list of program counters of the executed instructions up to the bug occurrence, all memory accesses done by each instruction (address and value) and the type of the access (read or write). Traces contain information about creation and propagation of all symbolic values and constraints on branches taken. Each branch instruction has a flag indicating whether it forked execution or not, thus enabling DDT to subsequently reconstruct an execution tree of the explored paths; each node in the tree corresponds to a machine state. Finally, DDT associates with each failed path a set of concrete inputs and system events (e.g., interrupts) that take the driver along that path. The inputs are derived from the symbolic state by solving the corresponding path constraints [16, 7].

A DDT trace has enough information to replay the bug in the DDT VM. Each trace starts from an initial state (a "hibernated" snapshot of the system) and contains the exact sequence of instructions and memory accesses leading to the crash or hang. The traces are self-contained and directly executable. The size of these traces rarely exceeds 1 MB per bug, and usually they are much smaller. We believe DDT traces can easily be adapted to work with existing VM replay tools [13, 29, 21].

DDT also post-processes these traces off-line, to produce a palatable error report. DDT reconstructs the tree of execution paths and, for each leaf state that triggered a bug, it unwinds the execution path by traversing the execution tree to the root. Then it presents the corresponding execution path to the developer. When driver source code is available, DDT-produced execution paths can be automatically mapped to source code lines and variables, to help developers better visualize the buggy behavior.

For bugs leading to crashes, it is also possible to extract a Windows crash dump that can be analyzed with

WinDbg [25]—since each execution state maintained by DDT is a complete snapshot of the system, this includes the disk where the OS saved the crash dump. It is also worth noting that DDT execution traces can help debuggers go backwards through the buggy execution.

In theory, DDT traces could be directly executed outside the VM (e.g., in a debugger) using a natively executing OS, since the traces constitute slices through the driver code. The problem, though, is that the physical hardware would need to be coerced into providing the exact same sequence of interrupts as in the trace—perhaps this could be done with a PCI-based FPGA board that plays back a trace of interrupts. Another challenge is providing the same input and return values to kernel calls made by the driver—here DDT could leverage existing hooking techniques [4, 18] to intercept and modify these calls during replay. Finally, replaying on a real machine would involve triggering asynchronous events at points equivalent to those saved in the traces [33].

3.6 Analyzing Bugs

Execution traces produced by DDT can also help understand the cause of a bug. For example, if an assertion of a symbolic condition failed, execution traces can identify on what symbolic values the condition depended, when during the execution were they created, why they were created, and what concrete assignment of symbolic values would cause the assertion to fail. An assertion, bad pointer access, or a call that crashes the kernel might depend indirectly on symbolic values, due to control flow-based dependencies; most such cases are also identifiable in the execution traces.

Based on device specifications provided by hardware vendors, one can decide whether a bug can only occur when a device malfunctions. Say a DDT symbolic device returned a value that eventually led to a bug; if the set of possible concrete values implied by the constraints on that symbolic read does not intersect the set of possible values indicated by the specification, then one can safely conclude that the observed behavior would not have occurred unless the hardware malfunctioned.

One could write tools to automate the analysis and classification of bugs found by DDT, even though doing this manually is not hard. They could provide both user-readable messages, like "driver crashes in low-memory situations," and detailed technical information, like "`AllocateMemory` failed at location pc_1 caused a null pointer dereference at some other location pc_2.

4 DDT Implementation

We now describe our implementation of a DDT prototype for Windows device drivers (§4.1), which can be used by both developers and consumers to test binary drivers before installing them. We also show how to trick Windows into accepting DDT's symbolic hardware (§4.2) and how to identify and exercise the drivers' entry points (§4.3). Although Windows-specific, these techniques can be ported to other platforms as well.

4.1 DDT for Microsoft Windows

DDT uses a modified QEMU [2] machine emulator together with a modified version of the Klee symbolic execution engine [6]. DDT can run a complete, unmodified, binary software stack, comprising Windows, the drivers to be tested, and all associated applications.

4.1.1 Doing VM-Based Symbolic Execution

QEMU is an open-source machine emulator that supports many different processor architectures, like x86, SPARC, ARM, PowerPC, and MIPS. It emulates the CPU, memory, and devices using dynamic binary translation. QEMU's support of multiple architectures makes DDT available to more than just x86-based platforms.

DDT embeds an adapted version of Klee. To symbolically execute a program, one first compiles it to LLVM bitcode [22], which Klee can then interpret. Klee employs various constraint solving optimizations and coverage heuristics, which make it a good match for DDT.

To use Klee, we extended QEMU's back-end to generate LLVM bitcode. QEMU translates basic blocks from the guest CPU instruction set to a QEMU-specific intermediate representation—we translate from this intermediate representation to LLVM on the fly. The generated LLVM bitcode can be directly interpreted by Klee.

QEMU and Klee have different representations of program state, which have to be kept separate yet synchronized. In QEMU, the state is composed of the virtual CPU, VM physical memory, and various virtual devices. We encapsulate this data in Klee memory objects, and modified QEMU to use Klee's routines to manipulate the VM's physical memory. Thus, whenever the state of the CPU is changed (e.g., register written) or a device is accessed (e.g., interrupt controller registers are set), both QEMU and Klee see it, and Klee can perform symbolic execution in a consistent environment.

Symbolic execution generates path constraints that also have to be synchronized. Since QEMU and Klee keep a synchronized CPU, device, and memory state, any write to the state by one of them will be reflected in the path constraints kept by Klee. For example, when symbolically executing driver code accesses concrete kernel memory, it sees data consistent with its own execution so far. Conversely, when concrete code attempts to access a symbolic memory location, that location is automatically concretized, and a corresponding constraint is added to the current path. Data written by concrete code is seen as concrete by symbolically running driver code.

4.1.2 Symbolic Execution of Driver Code

QEMU runs in a loop, continuously fetching guest code blocks, translating them, and running them on the host CPU or in Klee. When a basic block is fetched, DDT checks whether the program counter is inside the driver of interest or not. If yes, QEMU generates a block of LLVM code (or fetches the code from a translation cache) and passes it to Klee; otherwise, it generates x86 machine code and sends it to the host processor.

DDT monitors kernel code execution and parses kernel data structures to detect driver load attempts. DDT catches the execution of the OS code responsible for invoking the load entry point of device drivers. For example, on Windows XP SP3, DDT monitors attempts to execute code at address `0x805A3990`, then parses the stack to fetch the device object. If the name of the driver corresponds to the one being monitored, DDT further parses the corresponding data structures to retrieve the code and data segment locations of the driver. Parsing the data structures is done transparently, by probing the virtual address space, without causing any side effects (e.g., no page faults are induced).

When the driver is executed with symbolic inputs, DDT forks execution paths as it encounters conditional branches. Forking consists primarily of making a copy of the contents of the CPU, the memory, and the devices, to make it possible to resume the execution from that state at a later time. In other words, each execution state consists conceptually of a complete system snapshot.

4.1.3 Optimizing Symbolic Execution

Since symbolic execution can produce large execution trees (exponential in the number of branches), DDT implements various optimizations to handle the large number of states generated by Klee. Moreover, each state is big, consisting of the entire physical memory and of the various devices (such as the contents of the virtual disk).

DDT uses chained copy-on-write: instead of copying the entire state upon an execution fork, DDT creates an empty memory object containing a pointer to the parent object. All subsequent writes place their values in the empty object, while reads that cannot be resolved locally (i.e., do not "hit" in the object) are forwarded up to the parent. Since quick forking can lead to deep state hierarchies, we cache each resolved read in the leaf state with a pointer to the target memory object, in order to avoid traversing long chains of pointers through parent objects.

4.1.4 Symbolic Hardware

For PCI devices, the OS allocates resources (memory, I/O regions, and interrupt line) for the device, as required the device descriptor, prior to loading the driver, and then writes the addresses of allocated resources to the device's registers. From that point, the device continuously monitors all memory accesses on the memory and I/O buses; when an address matches its allocated address range, the device handles the access. In QEMU, such accesses are handled by read/write functions specific to a each virtual device. For DDT symbolic devices, the write functions discard their arguments, and the read functions always returns an unconstrained symbolic value. When DDT decides to inject a symbolic interrupt, it calls the corresponding QEMU function to assert the right interrupt assigned to the symbolic device by the OS.

The execution of the driver also depends on certain parts of the device descriptor, not just on the device memory and I/O registers. For example, the descriptor may contain a hardware revision number that triggers slightly different behavior in the driver. Unfortunately, the device descriptor is parsed by the OS when selecting the driver and allocating device resources, so DDT cannot just make it symbolic. Instead, as the device drivers always accesses the descriptor through kernel API functions, we use annotations to insert appropriately constrained symbolic results when the driver reads the descriptor.

4.2 Fooling the OS into Accepting Symbolic Devices

Hardware buses like PCI and USB support Plug-and-Play, which is a set of mechanisms that modern operating systems use to detect insertion and removal of devices. The bus interface notifies the OS of such events. When the OS detects the presence of a new device, it loads the corresponding driver. The right driver is selected by reading the vendor and device ID of the inserted device. If the driver is for a PCI device, it will typically need to read the rest of the descriptor, i.e., the size of the register space and various I/O ranges.

DDT provides a PCI descriptor for a *fake device* to trick the OS into loading the driver to be tested. The fake device is an empty "shell" consisting of a descriptor containing the vendor and device IDs, as well as resource information. The fake device itself does not implement any logic other than producing symbolic values for read requests. Support for USB is similar: a USB descriptor pointing to a "shell" device is passed to the code implementing the bus, causing the target driver to be loaded.

Hardware descriptors are simple and can be readily obtained. If the actual hardware is available, the descriptors can be read directly from it. If the hardware is not present, it is possible to extract the information from pub-

Tested Driver	**Size of Driver Binary File**	**Size of Driver Code Segment**	**Number of Functions in Driver**	**Number of Called Kernel Functions**	**Source Code Available ?**
Intel Pro/1000	168 KB	120 KB	525	84	No
Intel Pro/100 (DDK)	70 KB	61 KB	116	67	Yes
Intel 82801AA AC97	39 KB	26 KB	132	32	No
Ensoniq AudioPCI	37 KB	23 KB	216	54	No
AMD PCNet	35 KB	28 KB	78	51	No
RTL8029	18 KB	14 KB	48	37	No

Table 1: Characteristics of Windows drivers used to evaluate DDT.

lic databases of hardware supported on Linux. If this information is not available, it can be extracted from the driver itself. For example, Windows drivers come with a `.inf` file specifying the vendor and device IDs of the supported devices. The device resources (e.g., memory or interrupt lines) are not directly available in the `.inf` files, but can be inferred after the driver is loaded, by watching for attempts to register the I/O space using OS APIs. We are working on a technique to automatically determine this information directly from the driver.

4.3 Exercising Driver Entry Points

DDT must detect that the OS has loaded a driver, determine the driver's entry points, coerce the OS into invoking them, and then symbolically execute them.

DDT automatically detects a driver's entry points by monitoring attempts of the driver to register such entry points with the kernel. Drivers usually export only one entry point, specified in the driver binary's file header. Upon invocation by the kernel, this routine fills data structures with entry point information and calls a registration function (e.g., `NdisMRegisterMiniport` for network drivers). In a similar way, DDT intercepts the registration of interrupt handlers.

DDT uses Microsoft's Device Path Exerciser as a concrete workload generator to invoke the entry points of the drivers to be tested. Device Path Exerciser is shipped with the Windows Driver Kit [25] and can be configured to invoke the entry points of a driver in various ways, testing both normal and error situations.

Each invoked entry point is symbolically executed by DDT. To accomplish this, DDT returns symbolic values on hardware register reads and, hooks various functions to inject symbolic data. Since execution can fork on branches within the driver, the execution can return to the OS through many different paths. To save memory and time, DDT terminates paths based on user-configurable criteria (e.g., if the entry point returns with a failure).

DDT attempts to maximize driver coverage using pluggable heuristics modules. The default heuristic attempts to maximize basic block coverage, similar to the one used in EXE [7]. It maintains a global counter for each basic block, indicating how many times the block was executed. The heuristic selects for the next execution step the basic block with the smallest value. This avoids states that are stuck, for instance, in polling loops (typical of device drivers). Depending on the driver, it is possible to choose different heuristics dynamically.

DDT tests for concurrency bugs by injecting symbolic interrupts before and after each kernel function called by the driver. It asserts the virtual interrupt line, causing QEMU to interrupt the execution of the current code and to invoke the OS's interrupt handler. The injection of symbolic interrupts is activated as soon as the target driver registers an interrupt handler for the device.

Drivers may legitimately access the kernel's data structures, and this must be taken into account by DDT, to avoid false reports of unauthorized memory accesses. First, drivers access global kernel variables, which must be explicitly imported by the driver; DDT scans the corresponding section of the loaded binary and grants the driver access to them. Second, private kernel data may be accessed via inlined functions (for example, NDIS drivers use macros that access kernel-defined private data fields in the `NDIS_PACKET` data structure). DDT provides annotations for identifying such data structures.

5 Evaluation

We applied DDT to six mature Microsoft-certified drivers—DDT found 14 serious bugs (§5.1). We also measured code coverage, and found that DDT achieves good coverage within minutes (§5.2). All reported measurements were done on an Intel 2 GHz Xeon CPU using 4 GB of RAM.

5.1 Effectiveness in Finding Bugs

We used DDT to test four network drivers and two sound card drivers, which use different Windows kernel APIs and are written in both C and C++ (Table 1). All drivers are reasonably sized, using tens of API functions; DDT scales well in this regard, mainly due to the fact that it needs no kernel API models. Most of these drivers have been tested by Microsoft as part of the WHQL certification process [25] and have been in use for many years.

DDT found bugs in all drivers we tested: memory leaks, memory corruptions, segmentation faults, and race

Tested Driver	Bug Type	Description
RTL8029	Resource leak	Driver does not always call `NdisCloseConfiguration` when initialization fails
RTL8029	Memory corruption	Driver does not check the range for `MaximumMulticastList` registry parameter
RTL8029	Race condition	Interrupt arriving before timer initialization leads to BSOD
RTL8029	Segmentation fault	Crash when getting an unexpected OID in `QueryInformation`
RTL8029	Segmentation fault	Crash when getting an unexpected OID in `SetInformation`
AMD PCNet	Resource leak	Driver does not free memory allocated with `NdisAllocateMemoryWithTag`
AMD PCNet	Resource leak	Driver does not free packets and buffers on failed initialization
Ensoniq AudioPCI	Segmentation fault	Driver crashes when `ExAllocatePoolWithTag` returns NULL
Ensoniq AudioPCI	Segmentation fault	Driver crashes when `PcNewInterruptSync` fails
Ensoniq AudioPCI	Race condition	Race condition in the initialization routine
Ensoniq AudioPCI	Race condition	Various race conditions with interrupts while playing audio
Intel Pro/1000	Memory leak	Memory leak on failed initialization
Intel Pro/100 (DDK)	Kernel crash	`KeReleaseSpinLock` called from DPC routine
Intel 82801AA AC97	Race condition	During playback, the interrupt handler can cause a BSOD

Table 2: Summary of previously unknown bugs discovered by DDT.

conditions. A summary of these findings is shown in Table 2, which shows *all* bug warnings issued by DDT, not just a subset. In particular, we encountered no false positives during testing.

The first two columns of the table are a direct output from DDT. Additionally, DDT produced execution traces that we manually analyzed (as per §3.6) in order to produce the last column of the table, explaining each bug. The analyses took a maximum of 20 minutes per bug. Testing each driver took a maximum of 4 hours, and this time includes adding missing API annotations and occasional debugging of the DDT prototype.

From among all bugs found by DDT, only one was related to improper hardware behavior: it was a subtle race condition in the RTL8029 driver, occurring right after the driver registered its interrupt handler, but before it initialized the timer routine and enabled interrupts on the device. If the interrupt fires at this point, the interrupt handler calls a kernel function to which it passes an uninitialized timer descriptor, causing a kernel crash. From the execution traces produced by DDT it was clear that the bug occurred in the driver interrupt handler routine after issuing a symbolic interrupt during driver initialization. We checked the address of the interrupt control register in the device documentation; since the execution traces contained no writes to that register, we concluded that the crash occurred before the driver enabled interrupts.

At the same time, if the device malfunctions and this bug manifests in the field, it is hard to imagine a way in which it could be fixed based on bug reports. It is hard to find this kind of bugs using classic stress-testing tools, even with malfunctioning hardware, because the interrupt might not be triggered by the hardware at exactly the right moment.

Another interesting bug involved memory corruption after parsing parameters (obtained from the registry) in the RTL8029 driver. The driver does not do any bounds checking when reading the *MaximumMulticastList* parameter during initialization. Later, the value of this parameter is used as an index into a fixed-size array. If the parameter has a large (or negative) value, memory corruption ensues and leads to a subsequent kernel panic. This explanation was easily obtained by looking at the execution traces: a faulty memory read was shown at an address equal to the sum of the base address returned by the memory allocator plus an unconstrained symbolic value injected when reading the registry.

An example of a common kind of bug is the incorrect handling of out-of-memory conditions during driver initialization. In the RTL8029, AMD PCNet, and Intel Pro/1000 drivers, such conditions lead to resource leaks: when memory allocation fails, the drivers do not release all the resources that were already allocated (heap memory, packet buffers, configuration handlers, etc.). In the Ensoniq AudioPCI driver, failed memory allocation leads to a segmentation fault, because the driver checks whether the memory allocation failed, but later uses the returned null pointer on an error handling path, despite the fact that the initial check failed.

An example of incorrectly used kernel API functions is a bug in the Intel Pro/100 driver. In its DPC (deferred procedure call) routine, the driver uses the `NdisReleaseSpinLock` function instead of `NdisDprReleaseSpinLock` (as it should for spinlocks acquired using `NdisDprAcquireSpinLock`). This is specifically prohibited by Microsoft documentation and in certain conditions can lead to setting the IRQ level to the wrong value, resulting in a kernel hang or panic.

We tried to find these bugs with the Microsoft Driver Verifier [25] running the driver concretely, but did not find any of them. Furthermore, since Driver Verifier crashes by default on the first bug found, looking for the next bug would typically require first fixing the found bug. In contrast, DDT finds multiple bugs in one run.

To assess the influence that annotations have on DDT's effectiveness, we re-tested these drivers with all

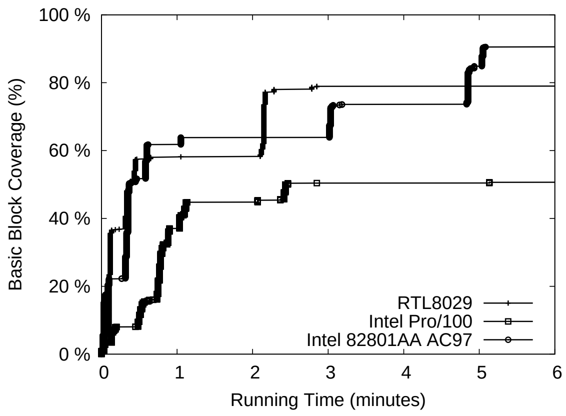

Figure 2: Relative coverage with time

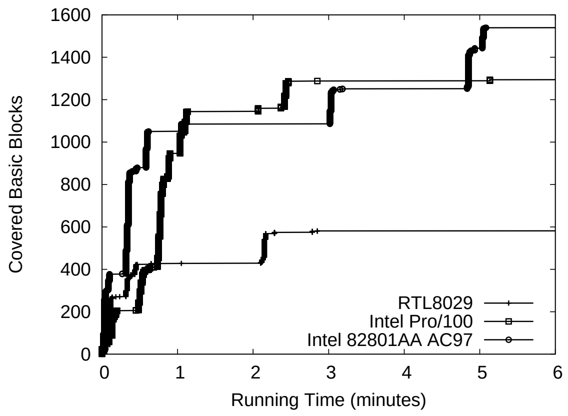

Figure 3: Absolute coverage with time

annotations turned off. We managed to reproduce all the race condition bugs, because their detection does not depend on the annotations. We also found the hardware-related bugs, cased by improper checks on hardware registers. However, removing the annotations resulted in decreased code coverage, so we did not find the memory leaks and the segmentation faults.

We initially wanted to compare DDT to the Microsoft SDV tool [1], a state-of-the-art static analysis tool for drivers. Since SDV requires source code, we used the Intel Pro/100 network card driver, whose source code appears in the Windows Drivers Development Kit. Unfortunately, we were not able to test this driver out-of-the-box using SDV, because the driver uses older versions of the NDIS API, that SDV cannot exercise. SDV also requires special entry point annotations in the source code, which were not present in the Intel Pro/100 driver. We resorted to comparing on the sample drivers shipped with SDV itself: SDV found the 8 sample bugs in 12 minutes, while DDT found all of them in 4 minutes.

We additionally injected several synthetic bugs in the sample driver (most of these hang the kernel): a deadlock, an out-of-order spinlock release, an extra release of a non-acquired spinlock, a “forgotten” unreleased spinlock, and a kernel call at the wrong IRQ level. SDV did not find the first 3 bugs, it found the last 2, and produced 1 false positive. DDT found all 5 bugs and no false positives in less than a third of the time that SDV ran.

We conclude that DDT can test drivers that existing tools cannot handle, and can find more subtle bugs in mature device drivers. In the next section, we evaluate the efficiency of DDT and assess its scalability.

5.2 Efficiency and Scalability

We evaluated DDT on drivers ranging in size from 18 KB to 168 KB. In Figure 2 we show how code coverage (as a fraction of total basic blocks) varied with time for a representative subset of the six drivers we tested. In Figure 3 we show absolute coverage in terms of number of basic blocks. We ran DDT until no more basic blocks were discovered for some amount of time. In all cases, a small number of minutes were sufficient to find the bugs we reported. For the network drivers, the workload consisted of sending one packet. For the audio drivers, we played a small sound file. DDT’s symbolic execution explored paths starting from the exercised entry points. For more complex drivers, workload can be generated with the Device Path Exerciser (described in §4).

DDT has reasonable memory requirements. While testing the drivers in Table 1, DDT used at most 4 GB of memory, which is the current prototype’s upper limit.

The coverage graphs show long flat periods of execution during which no new basic blocks are covered. These periods are delimited by the invocation of new entry points. The explanation is that the driver-loading phase triggers the execution of many new basic blocks, resulting in a first step. Then, more paths are exercised in it, without covering new blocks. Finally, the execution moves to another entry point, and so on. Eventually, no new entry points are exercised, and the curves flatten.

Overall, the results show that high coverage of binary drivers can be achieved automatically in just a few minutes. This suggests that DDT can be productively used even by end users on their home machines.

6 Discussion

Having seen that DDT is able to automatically find bugs in a reasonable amount of time, we now discuss some of DDT’s limitations (§6.1) and the tradeoffs involved in testing binary drivers instead of their source code (§6.2).

6.1 Limitations

DDT subsumes several powerful driver testing tools, but still has limitations, which arise both from our design choices, as well as from technical limitations of the building blocks we use in the DDT prototype.

DDT uses symbolic execution, which is subject to the path explosion problem [3]. In the worst case, the number of states is exponential in the number of covered branches, and this can lead to high memory consumption and long running times for very large drivers. Moreover, solving path constraints at each branch is CPU-intensive. This limits DDT's ability to achieve good coverage for large drivers. We are exploring ways to mitigate this problem by running symbolic execution in parallel [12], and we are developing techniques for trimming the large space of paths to be explored [10]. Any improvements in the scalability of symbolic execution automatically improve DDT's coverage for very large drivers.

Like any bug finding tool, DDT might have false negatives. There are two causes for this: not checking for a specific kind of bug, or not covering a path leading to the bug. Since DDT can reuse any existing dynamic bug finding tool (by running it inside the virtual machine along all explored paths) and can be extended with other types of checkers, we expect that DDT can evolve over time into a tool that achieves superior test completeness.

Since DDT does not use real hardware and knows little about its expected behavior, DDT may find bugs that can only be triggered by a malfunctioning device. Even though it has been argued that such cases must be handled in high-reliability drivers [19], for some domains this may be too much overhead. In such cases, these false positives can be weeded out by looking at the execution traces, or by adding device-specific annotations.

Some driver entry points are triggered only when certain conditions hold deep within the execution tree. For example, the `TransferData` entry point in an NDIS driver is typically called when the driver receives a packet *and* provides some look-ahead data from it to the kernel *and* the kernel finds a driver that claims that packet. Since the packet contains purely symbolic data, and is concretized randomly when the kernel reads it, the likelihood of invoking the required handler is low. Annotating the function transmitting the look-ahead data to the kernel can solve this problem.

While testing drivers with DDT can be completely automated, our current DDT prototype requires some manual effort. A developer must provide DDT with PCI device information for the driver's device, install the driver inside a VM, and configure Microsoft Driver Verifier and a concrete workload generator. Once DDT runs, its output is a list of bugs and corresponding execution traces; the developer can optionally analyze the execution traces to find the cause of the encountered bugs. Even though this limits DDT's immediate usefulness to end users, DDT can be used today by hardware vendors to test drivers before releasing them, by OS vendors to certify drivers, and by system integrators to test final products before deployment.

DDT does not yet support USB, AGP, and PCI-express devices, partly due to the lack of such support in QEMU. This limitation prevents DDT from loading the drivers, but can be overcome by extending QEMU.

Finally, DDT currently has only a 32-bit implementation. This prevents DDT from using more than 4 GB of memory, thus limiting the number of explored paths. Although we implemented various optimizations, like swapping out unnecessary states to disk, memory is eventually exhausted. We ported Klee to 64-bit architectures and contributed it to the Klee mainline; we intend to port DDT as well.

6.2 Source-Level vs. Binary-Level Testing

DDT is a binary-level testing tool, and this has both benefits and drawbacks.

A binary tool can test the end result of a complex build tool chain. Device drivers are built with special compilers and linked to specific libraries. A miscompilation (e.g., a wrong field alignment in the data structures), or linking problems (e.g., a wrong library version), can be more easily detected by a binary testing tool.

Binary drivers, however, have meager typing information. The only types used by binaries are integers and pointers, and it may be possible to infer some types by looking at the API functions for which we have parameter information. Nevertheless, it is hard to find type-related bugs that do not result in abnormal operations. For example, a cast of a color value from the framebuffer to the wrong size could result in incorrect colors. Such bugs are more easily detected by source code analyzers.

7 Related Work

Two main approaches have been taken to improve the safety and reliability of device drivers. Offline approaches, like testing, concentrate on finding bugs before the driver is shipped. However, thoroughly testing a driver to the point of guaranteeing the absence of bugs is still economically infeasible, so bugs frequently make their way to the field. Online approaches aim to protect systems from bugs that are missed during the testing phase, but typically at the expense of runtime overhead and/or modifications to the OS kernel.

Testing device drivers can be done statically or dynamically. For example, SLAM [1] statically checks the source code of a device driver for correct Windows API usage. It uses a form of model checking combined with an abstract representation of the source code, suitable for the properties to be checked. However, it is subject to false positives and false negatives stemming from incomplete and/or imprecise API models.

Microsoft provides various tools for stress-testing device drivers running in their real environment. For example, Driver Verifier [25] provides deep testing of running device drivers, but it can miss rarely executed code paths. DDT combines the power of both static and dynamic tools: it runs drivers in a real environment, and combines its own checks with those of the Driver Verifier. Moreover, DDT employs fully symbolic hardware, leading to a more thorough exploration.

When testing is not enough, it is possible to continuously monitor the drivers at runtime and provide information on the cause of the crashes. For example, Nooks [31] combines in-kernel wrapping and hardware-enforced protection domains to trap common faults and recover from them. Nooks works with existing drivers, but requires source code and incurs runtime overhead.

SFI [32] and XFI [14] use faster software isolation techniques and provide fine grained isolation of the drivers to protect the kernel from references outside their logical protection domain. However, it can only protect against memory failures and incurs runtime overhead. XFI can work on binary drivers but still requires debugging information for the binaries in order to reliably disassemble them. SafeDrive [35] uses developer provided annotations to enforce type-safety constraints and system invariants for kernel-mode drivers written in C. Finally, BGI [8] provides byte-granularity memory protection to sandbox kernel extensions. BGI was also able to find driver bugs that manifest when running the drivers with BGI isolation. However BGI also requires access to the source code and incurs runtime overhead.

Minix [17] explicitly isolate drivers by running them in distinct address spaces; this approach is suitable for microkernels. Vino [28] introduces an alternative OS design, which combines software fault isolation with a lightweight transactional system to protect against large classes of problems in kernel extensions.

The idea of replacing reads from hardware with symbolic values has been mentioned before [3]. With DDT, we introduce the new concept of fully symbolic hardware, which can interact both with concretely running OSes and with symbolically running device drivers. Fully symbolic hardware can also issue symbolic interrupts, enabling the testing of various interleavings of device driver code and interrupt handlers.

Selective symbolic execution was first introduced in [10] and later reused in [9]. DDT shares common ideas with these, but is also distinguished by several aspects.

First, reverse engineering of a driver with RevNIC does not require execution to be sound. For example, RevNIC overwrites with unconstrained symbolic values the concrete parameters passed by the OS to the driver. In contrast, since DDT is a testing tool, it requires the execution to be sound to avoid false positives. This introduces additional requirements on injection of symbolic values and on concretization. For example, the concrete packet size must be replaced by a symbolic value constrained not to be greater than the original value, to avoid buffer overflows.

Second, DDT introduces the use of lightweight API annotations to describe the interface between a driver and a kernel. Annotations encode developers' knowledge about a specific kernel API, and help improve code coverage as well as detect more logic bugs. Such annotations were not present in RevNIC.

Third, DDT mixes in-VM instrumentation (bug checking) with instrumentation from outside the VM. DDT can reuse existing bug-finding tools that run in the guest OS, extending these tools with symbolic execution to work on multiple paths.

Finally, during symbolic execution, RevNIC only gathers executed LLVM code and traces of device accesses. In contrast, DDT analyzes the execution in order to track the origin of symbolic values and control flow dependencies through the path leading to a bug. DDT generates annotated execution traces and input values that help developers reproduce and understand the bugs.

8 Conclusion

We presented DDT, a tool for testing closed-source binary device drivers against undesired behaviors, like race conditions, memory errors, and resource leaks. We evaluated DDT on six mature Windows drivers and found 14 serious bugs that can cause a system to freeze or crash.

DDT combines virtualization with selective symbolic execution to thoroughly exercise tested drivers. A set of modular dynamic checkers identify bug conditions and produce detailed, executable traces for every path that leads to a failure. We showed how these traces can be used to provide evidence of the found bugs, as well as help understand and fix them.

DDT does not require access to source code and needs no assistance from users, thus making it widely applicable. We envision DDT being used by IT staff responsible for the reliability and security of desktops and servers, by OS vendors and system integrators, as well as by consumers who wish to avoid running buggy drivers in their operating system kernels.

Acknowledgments

We are indebted to our shepherd, Paul Barham, the anonymous USENIX reviewers, and our EPFL colleagues for their great help in improving our paper.

References

[1] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *ACM SIGOPS/EuroSys European Conf. on Computer Systems*, 2006.

[2] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conf.*, 2005.

[3] P. Boonstoppel, C. Cadar, and D. R. Engler. RWset: Attacking path explosion in constraint-based test generation. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[4] P. P. Bungale and C.-K. Luk. PinOS: a programmable framework for whole-system dynamic instrumentation. In *Intl. Conf. on Virtual Execution Environments*, 2007.

[5] S. Butt, V. Ganapathy, M. M. Swift, and C.-C. Chang. Protecting commodity operating system kernels from vulnerable device drivers. In *Annual Computer Security Applications Conf.*, 2009.

[6] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symp. on Operating Systems Design and Implementation*, 2008.

[7] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Conf. on Computer and Communication Security*, 2006.

[8] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *Symp. on Operating Systems Principles*, 2009.

[9] V. Chipounov and G. Candea. Reverse engineering of binary device drivers with RevNIC. In *ACM SIGOPS/EuroSys European Conf. on Computer Systems*, 2010.

[10] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. Selective symbolic execution. In *Workshop on Hot Topics in Dependable Systems*, 2009.

[11] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. 2001.

[12] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea. Cloud9: A software testing service. In *Workshop on Large Scale Distributed Systems and Middleware*, 2009.

[13] G. W. Dunlap, D. Lucchetti, P. M. Chen, and M. Fetterman. Execution replay on multiprocessor virtual machines. In *Intl. Conf. on Virtual Execution Environments*, 2008.

[14] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Symp. on Operating Systems Design and Implementation*, 2006.

[15] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP kernel crash analysis. 2006.

[16] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Conf. on Programming Language Design and Implementation*, 2005.

[17] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Fault isolation for device drivers. In *Intl. Conf. on Dependable Systems and Networks*, 2009.

[18] G. Hoglund and J. Butler. *Rootkits: subverting the Windows Kernel*. Campus Press, 2006.

[19] A. Kadav, M. J. Renzelmann, and M. M. Swift. Tolerating hardware device failures in software. In *Symp. on Operating Systems Principles*, 2009.

[20] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 1976.

[21] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX Annual Technical Conf.*, 2005.

[22] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Intl. Symp. on Code Generation and Optimization*, 2004.

[23] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *Symp. on Operating Systems Design and Implementation*, 2000.

[24] Microsoft security advisory #944653: Vulnerability in Macrovision driver. http://www.microsoft.com/technet/security/advisory/944653.mspx.

[25] Microsoft. Windows logo program. http://www.microsoft.com/whdc.

[26] B. Murphy. Automating software failure reporting. *ACM Queue*, 2(8), 2004.

[27] V. Orgovan and M. Tricker. An introduction to driver quality. Microsoft Windows Hardware Engineering Conf., 2003.

[28] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Symp. on Operating Systems Design and Implementation*, 1996.

[29] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conf.*, 2004.

[30] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. *ACM Transactions on Computer Systems*, 24(4), 2006.

[31] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 23(1), 2005.

[32] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Symp. on Operating Systems Principles*, 1993.

[33] B. Xin, W. N. Sumner, and X. Zhang. Efficient program execution indexing. In *Conf. on Programming Language Design and Implementation*, 2008.

[34] J. Zhang. A path-based approach to the detection of infinite looping. In *Asia-Pacific Conf. on Quality Software*, 2001.

[35] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: safe and recoverable extensions using language-based techniques. In *Symp. on Operating Systems Design and Implementation*, 2006.