Thorough Static Analysis of Device Drivers

Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner

Microsoft Corporation

ABSTRACT

Bugs in kernel-level device drivers cause 85% of the system crashes in the Windows XP operating system [44]. One of the sources of these errors is the complexity of the Windows driver API itself: programmers must master a complex set of rules about how to use the driver API in order to create drivers that are good clients of the kernel. We have built a static analysis engine that finds API usage errors in C programs. The Static Driver Verifier tool (SDV) uses this engine to find kernel API usage errors in a driver. SDV includes models of the OS and the environment of the device driver, and over sixty API usage rules. SDV is intended to be used by driver developers "out of the box." Thus, it has stringent requirements: (1) complete automation with no input from the user; (2) a low rate of false errors. We discuss the techniques used in SDV to meet these requirements, and empirical results from running SDV on over one hundred Windows device drivers.

Categories and Subject Descriptors

D.2.4 [**Software**]: Software Engineering—*Program Verification*; D.4.5 [**Software**]: Operating Systems—*Reliability*

General Terms

Reliability, Verification

Keywords

Software model checking, formal verification

1. INTRODUCTION

Writing a robust device driver requires a great deal of expertise and precise understanding of how drivers are supposed to interact with the operating system or kernel. Testing a device driver is just as tricky. There are two main difficulties that typically limit the testability of device drivers:

Observability: It is difficult to determine when something goes wrong in the interaction between a driver and the kernel. In the Windows operating system there are a large number of kernel-level APIs, which gives rise to many ways in which a driver can misuse these APIs. Such errors rarely lead to immediate failures. Instead, the system is left in an inconsistent state, resulting in a crash or improper behavior at a later time. It would be useful to detect the driver error at the point where the root cause of the error happens.

Controllability/Coverage: Drivers that work correctly under normal circumstances can have subtle errors that appear only under rare and exceptional situations. Such cases can be hard to purposefully exercise. As a result, traditional testing techniques usually fail to provide high coverage through the driver's set of execution paths.

What makes these problems particularly important is the fact that, at least in the Windows operating system, device drivers are the defacto mechanism for efficiently adding basic functionality into the operating system. In Linux, *kernel modules* provide a similar facility. Software for virus protection, virtual machine emulation, performance monitoring, and HTTP are all typically implemented, in part, as Windows kernel-level device drivers.

For this reason a surprising number of developers across the world are, in effect, Windows kernel developers. In order for a kernel to execute correctly on a machine, the developers of the drivers and kernel modules installed on that machine must have all written their code to obey the kernel-level API usage rules. Furthermore, features such as plug-and-play, power management and asynchronous I/O all substantially enhance yet complicate the Windows driver model—making them a common source of driver errors.

We present a tool called SDV that uses static analysis to enhance both the observability and coverage of device driver testing. Increased observability is obtained by stating and checking rules about the proper use of kernel APIs. Increased coverage is provided by a combination of two techniques: (1) a hostile model of the driver's execution environment tests the driver in many stressful scenarios, such as operating system calls continually failing; (2) an analysis engine—called SLAM[1]—based on model checking and symbolic execution that simulates all possible behaviors of the

[1]We will refer to SLAM as SDV*'s analysis engine* throughout the remainder of the article

code. This analysis engine seeks to find all ways that a device driver can disobey a set of API usage rules. Violations that are found by the analysis engine are then presented as source-level error paths through the driver code.

The Driver Abstraction Challenge. It is SDV's goal to check that device drivers make proper use of the driver API. It is not SDV's goal to check that device drivers perform any useful function with respect to their intended feature. Our hypothesis is that the amount of state that needs to be tracked in order to make an accurate determination about whether or not a driver obeys an API usage rule is relatively small compared to the entire state of the driver. The challenge is to automatically separate the relevant state from the irrelevant state.

SDV automatically abstracts the C code of a device driver to a simpler form. We call this alternative program an *abstraction* of the original because it does not lose errors: any API usage rule violation that appears in the original code also appears in the abstraction. This abstraction then can be checked efficiently against the API usage rule, which can be encoded as a state machine.

The program abstraction is expressed as a *Boolean program*, which has all the control-flow constructs of C (including procedures and procedure calls) but only Boolean variables. These variables track the state of relevant Boolean expressions in the C program. SDV automatically constructs a Boolean program from a C program and a set of predicates (Boolean expressions) to be observed. SDV uses a symbolic model checking algorithm based on binary decision diagrams [11] to check if a Boolean program obeys an API usage rule.

To give a rough example, consider a driver with 100,000 lines of source code and complicated data structures. Suppose the API usage rule being checked is intended to verify that a particular spin lock is properly used. To check this rule, SDV constructs a Boolean program where at each line of the program it keeps track of the state of the spin lock (via one Boolean variable), which can either be in the locked or unlocked state. Thus, the Boolean program can have on the order of 200,000 states (100,000 lines, with two states per line), which is well within the limits of symbolic model checking.

However, if this abstraction process were to yield too many false errors, SDV would be ineffective. When SDV finds an error path in the Boolean program, it checks that same path in the original C program to determine if it is a true error path. If necessary, rather than report false error paths to the user, SDV refines the Boolean program (through the addition of new predicates) to eliminate false error paths. This three step process of abstraction, model checking and refinement is repeated until a feasible error path is found or a proof of correctness is found.

1.1 Results and Overview

This paper makes the following contributions:

- It presents a static analysis tool that is able to find all errors that a device driver (C program) may contain with respect to a well-defined set of API usage rules. SDV's analysis has the effect of searching all code paths. It uses abstraction to make the analysis tractable and iterative refinement to greatly reduce the number of false errors reported. This analysis process distinguishes SDV from other dataflow analysis tools that do not perform refinement and, as a result, may report many false errors.
- It presents our experience with developing an environment model to stress the driver under analysis and a set of rules that specify what it means for a driver to be a good client of the Windows Driver Model (WDM) API. The rules and models have been tuned over several years, resulting in an automatic tool that works "out of the box" on the developer's desktop and has a low rate of false errors.
- It presents the results of running SDV on 126 WDM drivers with over 60 rules and on 20 KDMF drivers (KDMF is a new driver API for kernel-level Windows drivers) with over 40 rules. These results show that the tool finds multiple errors in almost every driver. We have investigated a number of error reports produced by SDV together with the developers of these drivers. We have found that 75%-80% of the errors that we have investigated were acknowledged as real errors in the drivers by the developers. In practice we found that all of the false errors that are reported are due to inaccuracies either in the rules or in the environment model. We continually refine both the rules and the environment model when we notice such false errors.

The remainder of the paper is organized as follows. Section 2 discusses related work. Sections 3 through 5 present the core components of SDV: Section 3 presents the rules, Section 4 discusses SDV's environment models and Section 5 presents the architecture and workings of SDV's analysis engine using an example. Section 6 illustrates a real error found by SDV on a Windows parallel port driver. Section 7 presents the results of running SDV on over one hundred Windows device drivers. Section 8 discusses some of the limitations of the SDV tool and how it can be improved. Section 9 concludes the paper.

2. RELATED WORK

The testing and verification of systems code is mature research area in which many advances have been made over the years. For example, run-time testing tools that instrument checks into a binary or OS system calls have been successfully used together with test cases. Purify, for example, performs this analysis in order to find array bounds violations and errors related to the reading and freeing of memory. Another example is Driver Verifier (DV) which examines the actions of a Windows driver during execution. DV is able to find many of the most frequently occurring errors in a driver—and these errors can be extremely deep.

The drawback of concrete execution tools is they only find errors which can be demonstrated during execution on the particular machines in which the driver is being tested, and only under the scenarios that are explicitly tried. This limits the coverage of the analysis. In contrast, SDV uses techniques such as model checking and symbolic execution to systematically get high coverage.

Another approach is that of driver isolation, where the driver writer is not given as much responsibility for the system's stability. Current research in this area focuses on finding the right balance between system performance and stability. Drivers can sometimes be executed in user-space, or

```
state {
   enum { Unlocked=0, Locked=1 }
     state = Unlocked;
}

KeAcquireSpinLock.return {
  if (state == Locked)
    error();
  else
    state = Locked;
}

KeReleaseSpinLock.return {
  if (!(state == Locked))
    error();
  else
    state = Unlocked;
}
```

(a)

```
enum { Unlocked=0, Locked=1 }
  state = Unlocked;

void KeAcquireSpinLock_return() {
  if (state == Locked)
    error();
  else
    state = Locked;
}

void KeReleaseSpinLock_return() {
  if (!(state == Locked))
    error();
  else
    state = Unlocked;
}
```

(b)

```
bool b1 = false;

void KeAcquireSpinLock_return() {
  if (b1)
    error();
  else
    b1 = true;
}

void KeReleaseSpinLock_return() {
  if (!b1)
    error();
  else
    b1 = false;
}
```

(c)

Figure 1: (a) An API usage rule for spin locks, (b) its compilation into C code and (c) its corresponding Boolean program.

the operating system can sometimes provide a virtual execution environment that appears to be kernel-space, but offers more protection to the system from driver faults. Examples of this approach include Nooks [44] and Xen [32]. Tools like CCured [41] can also be used to provide a limited form of isolation.

Detecting errors at compile time also is an active area of current research. Tools in this area are based on theorem proving, type systems, program analysis, model checking, and combinations of these techniques.

Tools based on theorem proving, such as ESC [40] and ESC/java [31] compile a program to a verification condition and use a theorem prover to prove the verification condition. These tools typically require a user to annotate preconditions and post-conditions on functions, and in certain cases loop invariants on loops. By powering up type systems, we can encode certain kinds of errors as type errors, and use the type checker to detect these errors. Examples of such systems include the vault language [27] and the cqual type-system. The esp tool checks C code against state machine properties [26] using interprocedural dataflow analysis. The mops tool [14] uses push-down model checking essentially on a reduced interprocedural control-flow graph of the program to check for security errors on large systems. Abstract interpretation [23] is a generic framework for studying all such analyses. Instantiation of the framework requires a specific abstract domain to be chosen by the designers of the tool. Since the abstraction is conservative, all these tools are prone to reporting false errors. In specific domains such as numerically intensive programs, the abstract interpreter can be tuned to reduce false errors to manageable limits [24].

In addition to the above tools, heuristic static analysis tools that do not attempt to cover all paths, have also demonstrated significant value. The PREfix [12] and PREfast tools [39] perform heuristic analysis that does not cover all of the execution paths, but has reportedly found many errors in source code within Microsoft. Tools from the Meta Compilation project at Stanford use heuristic analyses [29, 15, 33] as well, and they have successfully found many errors in Linux. The false errors produced by these tools can be managed after the analysis using techniques such as statistical ranking.

SDV's analysis engine, called Slam [1-9,37], implements automatic iterative refinement based on error paths. This idea first appeared in [36], and more recently in [16]. Both efforts deal with finite state systems. In addition to Slam, other tools have been built to check safety properties of C programs using iterative refinement, notably Blast [35] and Magic [13]. In the published literature, these tools have been applied to some device drivers, but on a small scale. In principle, SDV could use any of these model checkers as its analysis engine. The contribution of SDV is the combination of its C analysis engine together with the large and polished set of rules and environment models that are specific to Windows drivers. The contribution of this paper is in the application of techniques from Slam and Blast at an industrial scale.

3. API USAGE RULES

SDV's analysis engine checks *temporal safety* properties of sequential C programs. Roughly stated, temporal safety properties are those properties whose violation is witnessed by a finite execution path (see [38] for a formal definition). A simple example of a safety property is that a lock should be alternatingly acquired and released. We encode temporal safety properties in a C-like language that allows the definition of a safety automaton [43, 45]. The automaton monitors the execution behavior of a program at the level of function calls and returns. The automaton can read (but not modify) the state of the C program that is visible at the function call/return interface, maintain a history, and signal when a bad state occurs.

An API usage rule describes a state machine and has two components: (1) a static set of *state variables*, described as a C structure, and (2) a set of *events* and state transitions on the events. The state variables can be of any C type, including integers and pointers. Figure 1(a) shows a rule describing the proper usage of spin locks. There is one state variable `locked` that is initialized to `0`. There are two events on which state transitions happen —returns of calls to the

```
state {
  enum {Unlocked, Locked} state = Unlocked;
} watch KeAcquireSpinLock.$1;

KeAcquireSpinLock.return [guard $1] {
    if ( state == Locked ) {
        error;
    } else {
        state = Locked;
    }
}

KeReleaseSpinLock.return [guard $1] {
    if ( state == Unlocked ) {
        error;
    } else {
        state = Unlocked;
    }
}
```

Figure 2: Locking rule with `watch` and `guard` annotations.

functions `KeAcquireSpinLock` and `KeReleaseSpinLock`. Erroneous sequences of calls to these functions results in the execution of the `error` statement.

In fact, Figure 1 shows a simplified version of the real spin lock rule. Figure 2 shows a more complete version of this rule, which ensures that the analysis engine doesn't get confused by calls to `KeAcquireSpinLock` and `KeReleaseSpinLock` that acquire and release locks on different objects. The rule exhibits two additional elements: watch points and guards. The `watch` annotation to the state structure instructs SDV to track the state machine for each unique pointer value that can arise as the first parameter of `KeAcquireSpinLock`. (That is, the effect of the `watch` statement is to track the state machine for each particular pointer value that can flow into the first parameter of `KeAcquireSpinLock`). The `guard` annotation on the events identifies which parameter corresponds to the pointer value being "watched". Combined together, the effect of these two annotations is to instruct SDV to check the locking rule on each unique pointer value in isolation.

SDV comes with over 60 API usage rules (properties), ranging from simple locking properties (such as given above) to complex properties dealing with completion routines, plug-and-play, and power management. Figure 3 summarizes some of these rules. For example, the rule `markingqueuedirps` checks that drivers mark an I/O request packet as pending (using `IoMarkIrpPending`) before queuing it. Another rule, `pnpsurpriseremove`, checks that drivers do not call `IoDetachDevice` or `IoDeleteDevice` when processing a plug-and-play I/O request packet with type *surprise removal*.

These rules are a product of more than three years of effort. Each rule was developed from a suggestion in the documentation on Windows device drivers, and then rewritten and refined based on candidate violations found in device driver code when using SDV. While the rules were difficult to define and refine, the cost of their development is now being amortized over the value of the errors that are found in each new driver.

4. OS ENVIRONMENT MODEL

```
NTSTATUS
IoAllocateAdapterChannel(
   ADAPTER_OBJECT * AdapterObject,
   DEVICE_OBJECT * DeviceObject,
   ULONG NumberOfMapRegisters,
   DRIVER_CONTROL * ExecutionRoutine,
   void * Context
   )
{
    ULONG choice = SdvMakeChoice();
    if (choice==0) {
        return STATUS_SUCCESS;
    } else {
        return STATUS_INSUFFICIENT_RESOURCES;
    }
}
```

Figure 4: SDV's model of the kernel routine `IoAllocateAdapterChannel`.

A device driver operates in the complex environment of the operating system and other drivers in the driver stack. Of course, for SDV to be usable it must analyze the source code of a driver without access to the source code of Windows or other drivers. For this reason SDV provides a model for the environment in which the driver is executing. This environment model is in the form of a C program and has two parts. The *harness* code simulates the operating system initializing and invoking the device driver (in various ways). The *stub* code provides the semantics for the kernel APIs that the driver might call.

The SDV environment model is quite hostile to the device driver under analysis. The harness probes the driver in many different ways and the stubs simulate the kernel behaving in both successful and failing modes. A key way we make the environment model hostile to the driver is through the introduction of non-deterministic behavior into the harness and stubs. This non-determinism simulates the kernel behaving in many unexpected ways, which is important for probing error paths in the driver. It is exactly these paths that are hard to cover with testing. The combination of non-determinism (in the model) with static analysis and symbolic execution (in the analysis engine) achieves the effect of covering all paths in the driver.

Figure 4 shows SDV's stub that overapproximates the meaning of the function function `IoAllocateAdapterChannel`. This stub encodes the possibilities that the procedure could return either a success or failure status. The analysis considers both possibilities at every call to this function. Both possibilities are considered due to SDV's special treatment of `SdvMakeChoice`. Each call to `SdvMakeChoice` returns a fresh symbolic (unknown) integer and assigns it into the variable `choice`. As a result, `IoAllocateAdapterChannel` will non-deterministically either return `STATUS_SUCCESS` or `STATUS_INSUFFICIENT_RESOURCES`, depending on the value of `choice`.

As mentioned before, the harness is the piece of C code that mimics the operating system initializing and invoking a driver. SDV associates one of two harnesses with each API usage rule. SDV's simple harness simulates the effect of all of the following possible events:

- calling any of the driver's dispatch routines
- calling the driver's `StartIo` routine

Rule	Summary
adddevice	Checks that a driver's AddDevice routine calls certain key APIs.
cancelspinlock	Checks that cancel spinlocks are locked and unlocked in strict alternation.
criticalregions	Checks for certian common problems when using critical regions.
danglingdeviceobjref	Checks that the driver calls `ObDereferenceObject` after calling `IoGetAttachedDeviceReference`.
doublecompletion	Checks that drivers do not complete an I/O request packet twice with `IoCompleteRequest`.
exclusiveresourceaccess	Checks for common problems with exclusive resource access.
forwardedatbadirql	Checks that I/O request packets that are forwarded to other drivers at the wrong interrupt request level
irpprocessingcomplete	Checks that dispatch routines completely process I/O request packets.
irql*	Many rules checking that functions are called at correct levels of interrupt request level.
lowerdriverreturn	Checks that, if a driver calls another driver that is lower in the stack, then the dispatch routine returns the same status that was returned by the lower driver.
markingqueuedirps	Checks that drivers mark I/O request packets as pending while queuing them.
markirppending	Checks that returns of `STATUS_PENDING` and `IoMarkIrpPending` are correlated.
pendedcompletedrequest	Checks that drivers do not return `STATUS_PENDING` if `IoCompleteRequest` has been called.
pnpirpcompletion	Checks that plug-and-play I/O request packets are passed on to the lower driver in the stack if one exists.
pnpsamedeviceobject	Checks that `IoAttachDeviceToDeviceStack` is called with an appropriate device object.
pnpsurpriseremove	Ensure that drivers do not detach or delete on `IRP_MN_SURPRISE_REMOVAL` I/O request packets.
queuedspinlock	Checks that queued spinlocks are locked and unlocked in strict alternation.
spinlock	Checks that spinlocks are locked and unlocked in strict alternation.
spinlocksafe	Checks for specific deadlock cases with spinlocks
startiocancel	Checks for cancellation races.
startiorecursion	Checks for potential recursion in StartIo routines.
targetrelationneedsref	Checks that dispatch routine call `ObReferenceObject` on pointers returned by another dispatch routine from a `TargetRelation` plug-and-play I/O request packet.
wmicomplete	Checks that dispatch routines do not return without completing a WMI I/O request packet.
wmiforward	Checks that dispatch routines do not return without forwarding WMI I/O request packets with disposition IrpForward.

Figure 3: Summaries of some of the API usage rules included with SDV.

- executing any deferred procedure calls
- executing any interrupt service routines

The routines are given symbolic inputs and arbitrary initial states. In other words, SDV is effectively asking: does an API usage rule hold for any of the available dispatch routines when called on any input request?; does a rule hold for all interrupt service routines if they are called from any state?; etc. If a driver passes a rule using this harness, the result is quite strong; it is valid regardless of the state of the system before or after the execution.

However, for some rules the correctness of a driver will depend on an event occurring in the driver's `DriverEntry` or `AddDevice` routines, or even the plug-and-play dispatch routine when invoked on an I/O request packet with type `IRP_MN_START_DEVICE`. In these cases SDV uses a more complicated harness which executes the driver symbolically with respect to the following events:

- The driver's `DriverEntry` routine, which initializes the driver's data structures, and then
- the driver's `AddDevice` routine, which adds the device and driver to their respective stacks, and then
- the driver's plug-and-play dispatch routine with an `IRP_MN_START_DEVICE` I/O request packet (to start the device), and then
- any dispatch routine, or deferred procedure call, or interrupt service routine, or the driver's `StartIo` function, and then
- the driver's plug-and-play dispatch routine with an `IRP_MN_REMOVE_DEVICE` I/O request packet (to simulate the device being removed from the computer) , and finally
- the driver's `Unload` routine, which the operating system would call after a device remove event.

This harness leads to fewer false errors being reported but results in increased analysis times.

5. ANALYSIS ENGINE

Figure 5 shows the architecture of SDV's analysis engine. SDV uses a technique called *counterexample-guided abstraction refinement* to automatically search for an abstract model of the original program which is sufficiently precise in order to prove the program's correctness with respect to an API usage rule or find a true error. The key idea is to find the C program state that is relevant to the rule being checked, and to discard the rest. The details of this process were published by the authors in previous papers. Here, we give just the briefest overview of the technique, by application to a small code example.

Figure 6(a) presents a sample of (simplified) C code from a PCI device driver that processes I/O request packets. We apply SDV to check if the code in Figure 6(a) obeys the locking API usage rule of Figure 1(a).

SDV first compiles the API usage rule into a set of C procedures (see Figure 1(b)), one for each event named in the API usage rule. SDV also performs a pointer analysis [25] on the program, which builds a graph representing a static overapproximation of the possible pointer alaising relationships between expressions occuring in the program. Function pointers are compiled away using explicit calls to the functions that appear in the aliasing graph. This graph is kept around, as it is used throughout SDV's analysis.

For each procedure p mentioned in the API usage rule, SDV finds all calls to procedure p in the code and instruments the code to call the appropriate procedure of Fig-

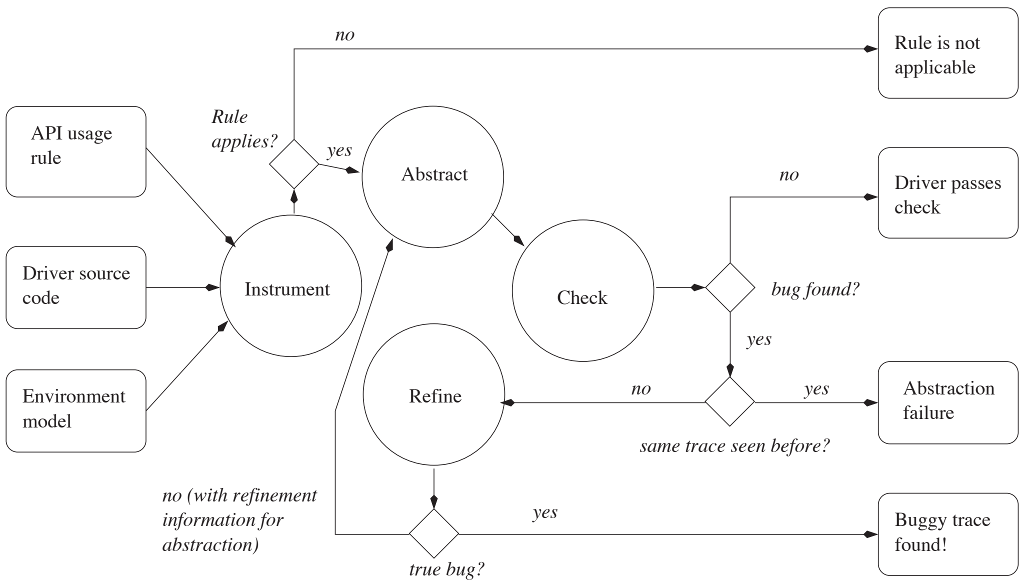

Figure 5: SDV's analysis engine's architecture

ure 1(b). Figure 6(b) shows the instrumented version of the code from Figure 6(a). Note that calls to the appropriate functions (from Figure 1(b)) are introduced at labels `A`, `B`, and `C` in Figure 6(b).

This serves to convert the API usage checking problem into a reachability problem: the function `error()` is called by the composite program (driver + rule + environment) if and only if the device driver violates the API usage rule. SDV's task is then to check that `error()` is not reachable in the composite program. If, during the instrumentation step, we discover that no event that triggers a call to `error()` can be instrumented, then we report that the rule is not applicable to the driver.

Otherwise, SDV passes the instrumented program to the abstraction module, called ABSTRACT. ABSTRACT automatically constructs a Boolean program abstraction of the original program with respect to a finite set of predicates. The set of initial predicates are those appearing in the C code of the API usage rule. In our example, this set of predicates consists of the single predicate `(state==Locked)` (as the other predicate is simply the negation of this predicate).

First, let's consider the translation of the C code in Figure 1(b) to the Boolean program code in Figure 1(c). Because `state` is a global variable and we wish to track the state of the predicate `(state==Locked)`, ABSTRACT introduces a global Boolean variable `b1` to track this predicate. The variable is initialized to **false** because the variable `state` is initialize to `Unlocked`. The translation of the other statements is straightforward, as the `state` variable already is acting as a Boolean variable. (We will see a more complicated example of abstraction soon.) The predicates `(state ==Locked)` and `!(state==Locked)` are translated to `(b1)` and `(!b1)`, as expected.

Now, let's consider the translation of the C code of Figure 6(b) to the Boolean program code of Figure 6(c). Note that many of the assignment statements in the `example` C procedure are abstracted to empty statements in the Boolean program. The ABSTRACT tool uses the points-to analysis to determine whether or not an assignment statement through a pointer dereference can affect the predicate `(state == Locked)`. The points-to analysis of the C program shows that no location in the `example` procedure can alias the address of the global `state` variable. Therefore, none of the assignment statements in the `example` procedure can affect the value of the predicate `(state==Locked)`. Furthermore: a side-effect analysis shows that none of the procedures, with the exception of `KeAcquireSpinLock_return` and `KeReleaseSpinLock_return`, can modify the variable `state`, so calls to these procedures are eliminated.

Finally, because none of the conditionals in the `example` are related to the `state` variable, they are replaced with calls to the `SdvMakeChoice` function (which non-deterministically returns an integer value). As a result of this abstraction of conditionals, the Boolean program will have more behaviors (feasible execution paths) than the original C program.

Once a Boolean program is constructed, SDV's CHECK module exhaustively explores all possible states of the Boolean program and checks whether the model can ever reach the `error` procedure. In the Boolean program of Figure 6(c), there are many execution paths leading to the `error` procedure. CHECK outputs a shortest error path which executes the function `KeAcquireSpinLock_return` twice in a row without an intervening call to `KeReleaseSpinLock_return`. This is possible because all the conditions in the procedure `example` have been abstracted to call `SdvMakeChoice`.

Because the C program and the Boolean program abstractions have identical control-flow graphs, the error path in the Boolean program also is a path of the C program. Now, this path may or may not be a feasible execution path of the original program. The REFINE module takes a C program and a potential error path as an input. It then uses verification condition generation to determine if the path is feasible. The answer may be "yes" or "no" or "don't know" (since this problem is undecidable, in general). If the answer is "yes" or "don't know" then SDV displays the path in the original driver code using a GUI that is similar to a visual

```
void example() {
 do {
  KeAcquireSpinLock();

  nPacketsOld = nPackets;
  req = devExt->WLHV;
  if(req && req->status){
    devExt->WLHV = req->Next;
    KeReleaseSpinLock();

    irp = req->irp;
    if(req->status > 0){
      irp->IoS.Status = SUCCESS;
      irp->IoS.Info = req->Status;
    } else {
      irp->IoS.Status = FAIL;
      irp->IoS.Info = req->Status;
    }
    SmartDevFreeBlock(req);
    IoCompleteRequest(irp);
    nPackets++;
  }
 } while(nPackets!=nPacketsOld);
 KeReleaseSpinLock();

}
```

(a)

```
void example() {
   do {
    KeAcquireSpinLock();
A:  KeAcquireSpinLock_return();
    nPacketsOld = nPackets;
    req = devExt->WLHV;
    if(req && req->status){
      devExt->WLHV = req->Next;
      KeReleaseSpinLock();
B:    KeReleaseSpinLock_return();
      irp = req->irp;
      if(req->status > 0){
        irp->IoS.Status = SUCCESS;
        irp->IoS.Info = req->Status;
      } else {
        irp->IoS.Status = FAIL;
        irp->IoS.Info = req->Status;
      }
      SmartDevFreeBlock(req);
      IoCompleteRequest(irp);
      nPackets++;
    }
   } while(nPackets!=nPacketsOld);
   KeReleaseSpinLock();
C: KeReleaseSpinLock_return();
 }
```

(b)

```
     void example() {
      do {
       ;
A:     KeAcquireSpinLock_return();
       ;
       ;
       if (SdvMakeChoice()) then
         ;
         ;
B:       KeReleaseSpinLock_return();
         ;
         if (SdvMakeChoice()) {
           ;
           ;
         } else {
           ;
           ;
         }
         ;
         ;
         ;
      }
      while (SdvMakeChoice());
      ;
C:    KeReleaseSpinLock_return();
     }
```

(c)

Figure 6: (a) A sample of device driver code P, (b) instrumented code P' that checks proper use of spin locks, and (c) initial Boolean program.

debugger.

Consider the (unique) execution path through the code in Figure 6(a) that executes `KeAcquireSpinLock` twice without executing an intervening call to `KeReleaseSpinLock`. Refine detects that the path is infeasible in the original program and generates the predicate `(nPackets!=npacketsOld)` as the explanation for the infeasibility. This is because the assignment of `nPacketsOld` to `nPackets` in the path makes the predicate `(nPackets!=npacketsOld)` false. Since the path does not contain the assignment statement `nPackets++` (since the path avoids the call to `KeReleaseSpinLock` inside the loop), the predicate still will be false at the end of the loop. However, in order to reach the second call to `KeAcquireSpinLock`, the path requires the predicate `(nPackets != npacketsOld)` to be true. This contradiction is easily detected using symbolic analysis.

Now, Abstract constructs the second Boolean program (see Figure 7), which has a new Boolean variable `b2` to track the state of the predicate `(nPackets!=npacketsOld)`. The conditional of the **do-while** is refined from `SdvMakeChoice()` to `(b2)` and two assignment statements now appear in the Boolean program. The first assignment statement is `b2=false` which captures the effect of the statement `nPacketsOld = nPackets` on the predicate of interest. The increment statement `nPackets++` translates to the statement

```
b2=!b2?true:SdvMakeChoice()
```

which captures the fact that if `nPacketsOld!=nPackets` is false before the assignment statement `nPackets++` then the predicate `nPacketsOld!=nPackets` must be true afterwards.

Executing the Check module on the refined Boolean program establishes that it cannot execute the `error` function. That is, all false error paths have been eliminated and no true error paths have been found. Put another way, SDV has established that the lock is held at the end of the **do-while** loop if and only if `nPacketsOld==nPackets`. So, if the lock is held then the loop terminates and `KeReleaseSpinLock` is called. Otherwise, the lock is not held and the loop iterates, calling `KeAcquireSpinLock` once more.

6. AN EXAMPLE ERROR FROM THE WINDOWS PARALLEL PORT DRIVER

The previous section showed how SDV can validate that a piece of code obeys an API usage rule. This section shows a real error in a shipping Windows device driver that was not found until SDV was applied to the driver. The parallel port device driver used in Windows XP is a variation of a device driver that was originally developed for Windows NT (on which Windows XP is based). This device driver is available as a sample in the Windows device driver development kit (DDK). It consists of 24536 lines of C code. The error was introduced when the device driver was updated to support plug-and-play. The error survived code reviews and extensive testing and was not found until SDV was applied to the driver.

The error was found when checking a SDV rule called `doublecompletion` which ensures that device driver dispatch routines do not call the kernel-level API `IoCompleteRequest` more than once on the same I/O request packet pointer. The meaning of `IoCompleteRequest` is akin to `free` in C. This function frees up the space pointed to by a pointer to a request packet. This space may be re-allocated and passed to another thread in the system. For this reason, calling `IoCompleteRequest` again on the same parameter I/O request packet can have disastrous consequences to the sys-

```
    void example() {
     do {
      ;
A:    KeAcquireSpinLock_return();
      b2 = false;
      ;
      if (SdvMakeChoice()) then
        ;
        ;
B:      KeReleaseSpinLock_return();
        ;
        if (SdvMakeChoice()) {
          ;
          ;
        } else {
          ;
          ;
        }
        ;
        ;
        b2 = !b2 ? true : SdvMakeChoice();
      }
     } while (b2);
     ;
C:   KeReleaseSpinLock_return();
    }
```

Figure 7: The refined Boolean program.

```
state { bool CompletionAlreadyCalled = 0; }

IoCompleteRequest.entry
{
  if (SdvHarnessIrp==$1) {
    if (CompletionAlreadyCalled) { error(); }
    else { CompletionAlreadyCalled = 1; }
  }
}
```

Figure 8: SDV rule checking for multiple calls to `IoCompleteRequest` on request packets passed to dispatch routines. This is a simplified version of the `doublecompletion` in SDV distribution.

tem's stability.

Figure 8 displays a simplified version of the rule. This rule is then checked over each dispatch routine found in the device driver, where the first parameter to the dispatch routine is a global variable defined in the harness as `SdvHarnessIrp`. The rule defines one event: *the calling of the kernel function* `IoCompleteRequest`. During SDV's analysis, if it is exploring a path through which there are calls to the function `IoCompleteRequest`, then the code presented with this event in Figure 8 will be executed.

In order to demonstrate this example error we have included the relevant code from the driver in Figures 9, 10, and 11. The key steps in the path found by SDV are as follows:

- The environment (the OS, or possibly another driver) calls the parallel port device driver's *close* dispatch routine, which is called `PptDispatchClose` (Fig.9, at line 134). The variable `Irp` is a pointer to the I/O request packet on which the driver should not repeatedly call `IoCompleteRequest`.
- `PptDispatchClose` calls `PptFdoClose` (Fig. 10, line 4).
- `PptFdoClose` enters the conditional statement at line 19 in Fig. 10 and calls `P4CompleteRequest` (Fig. 11, line 1775). `P4CompleteRequest` calls the kernel API `IoCompleteRequest` on a pointer which aliases the value of `Irp` from Figure 9 and returns to the call site.
- `PptFdoClose` (Fig. 10) leaves the conditional statement via the `goto` on line 26. At line 63, `PptFdoClose` calls `P4CompleteRequestReleaseRemLock` (Fig. 11, line 1790).
- `P4CompleteRequestReleaseRemLock` calls the function `P4CompleteRequest`, which makes the second call to `IoCompleteRequest` on a pointer that aliases `Irp` from Figure 9.

```
133 NTSTATUS
134 PptDispatchClose(PDEVICE_OBJECT DevObj,PIRP Irp) {
135    PFDO_EXTENSION fdx = DevObj->DeviceExtension;
136    P5TraceIrpArrival( DevObj, Irp );
137    if( DevTypeFdo == fdx->DevType ) {
138        return PptFdoClose( DevObj, Irp );
139    } else {
140        return PptPdoClose( DevObj, Irp );
141    }
142 }
```

Figure 9: Source code from dispatchRedirect.c in Parallel port device driver

Why wasn't this error previously detected by concrete execution tools or static analysis tools? Triggering it in real life requires putting the parallel port device driver into a state where it is handling a *close* request from the operating system while the user simultaneously physically removes the parallel port from the computer (via the removal of a laptop from a docking station, for example). This scenario is difficult to realize via testing. Furthermore, the static analysis required to find this error involves interprocedural program analysis and careful tracking of pointer relationships (to ensure that `IoCompleteRequest` is called twice on the same pointer value)—without this tracking a static analysis will report too many false errors.

7. EXPERIMENTAL RESULTS

7.1 Errors Found

We first present the results of applying SDV to 126 WDM drivers. This sample includes 26 DDK samples (see Figure 12) and 100 other kernel-mode drivers obtained from various sources (ranging in size from 48 to 130,000 lines of code with an average size of 12,000 lines of code). The set includes device drivers for basic ports, storage, USB, 1394-interface, mouse, keyboard, PCI battery and file system filters. These drivers were verified together with DLLs (so called "export drivers") they utilize. A total of twenty DLLs were involved. All these 126 drivers have been in use for many years. They are very well tested and have been code reviewed by Windows kernel experts. Additionally, the sources of the 26 DDK sample drivers have been open and available to anyone in the world for over five years. Thus, we did not expect to find many errors in these drivers.

```
NTSTATUS
PptFdoClose(
    IN  PDEVICE_OBJECT  DeviceObject,
    IN  PIRP            Irp
    )
{
    PFDO_EXTENSION   fdx = DeviceObject->DeviceExtension;
    NTSTATUS            status;

    PAGED_CODE();

    //
    // Verify that our device has not been SUPRISE_REMOVED. Generally
    //   only parallel ports on hot-plug busses (e.g., PCMCIA) and
    //   parallel ports in docking stations will be surprise removed.
    //
    if( fdx->PnpState & PPT_DEVICE_SURPRISE_REMOVED ) {
        //
        // Our device has been SURPRISE removed, but since this is only
        //   a CLOSE, SUCCEED anyway.
        //
        status = P4CompleteRequest( Irp, STATUS_SUCCESS, 0 );

        goto target_exit;
    }


    //
    // Try to acquire RemoveLock to prevent the device object from going
    //   away while we're using it.
    //
    status = PptAcquireRemoveLock(&fdx->RemoveLock, Irp);
    if( !NT_SUCCESS( status ) ) {
    // Our device has been removed, but since this is only a CLOSE .....
        status = STATUS_SUCCESS;
        goto target_exit;
    }

    //
    // We have the RemoveLock
    //

    ExAcquireFastMutex(&fdx->OpenCloseMutex);
    if( fdx->OpenCloseRefCount > 0 ) {
        //
        // prevent rollover -  strange as it may seem, it is perfectly
        //   legal for us to receive more closes than creates - this
        //   info came directly from Mr. PnP himself
        //
        if( ((LONG)InterlockedDecrement(&fdx->OpenCloseRefCount)) < 0 ) {
            // handle underflow
            InterlockedIncrement(&fdx->OpenCloseRefCount);
        }
    }
    ExReleaseFastMutex(&fdx->OpenCloseMutex);

target_exit:

    DD((PCE)fdx,DDT,"PptFdoClose - ........");

    return P4CompleteRequestReleaseRemLock( Irp, STATUS_SUCCESS, 0,
                                            &fdx->RemoveLock );
}

```

Figure 10: Source code from fdoClose.c in Parallel port device driver

On the 126 WDM drivers, SDV reported 206 defects, of which we have carefully investigated 65 to date. We double checked the results of our investigation with the developers who own and maintain the driver code. Of the 65 defects, 53 were true errors and 12 were false errors. Of the 60 rules packaged into SDV, all the defects were found from 40 rules. The other 20 rules are able to find injected defects but did not find any defects in the drivers analyzed.

We also have developed over 40 rules for a new driver API called Kernel-Mode Driver Framework (KMDF). KMDF implements the fundamental features required for kernel mode drivers, including complete support for plug-and-play, power management, I/O queues, DMA, and synchronization. Rule development for SDV influenced some KMDF design decisions, helped clarify coding patterns for drivers that KMDF is promoting, and made those design decisions precise by specifying them as API usage rules. For example, by writing rules that check request completion and cancellation of requests in the driver, some inconsistencies and ambiguities in the design have been discovered and corrected.

We applied SDV to 20 KMDF sample drivers, including a disk driver, a serial device driver that supports power management, as well as drivers for 1394, mouse, keyboard, PCI, modem and video. The sample drivers were written by the team that is developing KMDF by converting existing DDK samples. The code for the samples was reviewed by several independent experts. On these 20 drivers, SDV reported 18 defects, all carefully investigated together with the driver owners. Out of 18 defects found, 12 have been confirmed to be real errors in the drivers and have been corrected.

```

NTSTATUS
P4CompleteRequest(
    IN PIRP       Irp,
    IN NTSTATUS   Status,
    IN ULONG_PTR  Information
    )
{
    P5TraceIrpCompletion(Irp);
    Irp->IoStatus.Status      = Status;
    Irp->IoStatus.Information = Information;
    IoCompleteRequest(Irp,IO_NO_INCREMENT);
    return Status;
}


NTSTATUS
P4CompleteRequestReleaseRemLock(
    IN PIRP             Irp,
    IN NTSTATUS         Status,
    IN ULONG_PTR        Information,
    IN PIO_REMOVE_LOCK  RemLock
    )
{
    P4CompleteRequest(Irp,Status,Information);
    PptReleaseRemoveLock(RemLock,Irp);
    return Status;
}
```

Figure 11: Source code from util.c in Parallel port device driver

7.1.1 *True Errors*

The following is a brief summary of some of the true errors that SDV found in the 126 device drivers:

- In one particular path the device driver is marking I/O request packet pending with a kernel API, but is forgetting to also mark it pending by setting a flag in the data structure (the value of the flag is checked at the end of the dispatch routine by the SDV rule).
- The driver's dispatch routine is returning the return value `STATUS_PENDING` but has declared the I/O request packet as *completed* with `IoCompleteRequest`
- The driver is calling `IoStartNextPacket` from within its *StartIo* routine, which can lead to recursion that exceeds the stack space.
- Early in the execution the device driver is calling an API that can raise the *interrupt request level* of the thread, and then (much later) is calling another kernel API that should not be called when the *interrupt request level* is raised due to the fact that it touches paged data.
- `IoCompleteRequest` is being called while holding a spinlock, which can cause deadlock.

Driver	Lines of Code
src/vdd/dosioctl/krnldrvr	304
src/general/tracedrv/tracedrv	337
src/general/ioctl/sys	556
src/input/moufiltr	678
src/general/cancel/sys	702
src/input/kbfiltr	753
src/general/cancel/startio	760
src/general/event/sys	760
src/kernel/mca/imca/sys	803
src/general/toaster/toastmon	1010
src/wdm/1394/driver/1394diag	1923
src/wdm/1394/driver/1394vdev	1958
src/storage/filters/diskperf	2110
src/network/modem/fakemodem	2324
src/wdm/hid/gameenum	2797
src/general/toaster/bus	3633
src/kernel/serenum	4430
src/general/toaster/func	4755
src/input/mouclass	5042
src/storage/fdc/flpydisk	5074
src/input/kbdclass	5316
src/input/mouser	5476
src/storage/fdc/fdc	7101
src/input/pnpi8042/daytona	15398
src/kernel/serial	23197
src/kernel/parport	24536

Figure 12: The 26 drivers from the Windows DDK.

- The driver is detaching a device object from the device stack when handling a `IRP_MN_SURPRISE_REMOVAL` I/O request packet.
- Upon a driver exiting, an acquired resource is not released (for example, the rules `ZwRegistryOpen` and `CancelSpinLock` found such errors).

Figure 13 provides some details about the true errors that were found in the 26 DDK sample drivers. In the figure, "# functions" is the number of C functions from the device driver and OS environment model that occur in the path; "# steps" indicates the number of assignments, conditional checks, function calls or function returns that occured in the error path. These numbers show that the error paths are interprocedural in nature, spanning a good number of procedures in both the driver and OS model.

7.1.2 *False Errors*

The three most common causes for false errors found by SDV are deficiencies in: (1) the C model of the Windows kernel routines; (2) the API usage rules; (3) the harness that calls the dispatch routines. Few false errors have been attributed to SDV's analysis engine.

Let us address each of these problem areas in turn. First, our C model of the kernel is written by hand rather than derived automatically from analysis of the kernel. This model has been kept as simple as possible for the set of rules SDV checks. However, sometimes we abstract away too much from the state of the kernel. As a result, there may be correlations between two calls to the kernel that SDV misses.

Error ID	# functions	# steps	runtime (s)
1	16	94	73.332
2	19	134	15.074
3	12	73	69.259
4	9	67	5.515
5	6	29	5.187
6	30	212	426.612
7	47	344	985.443
8	6	51	36.216
9	14	105	12.190
10	13	99	11.909
11	15	110	12.393
12	26	137	69.126
13	14	115	12.500
14	14	141	573.742
15	16	123	61.736
16	15	132	55.626
17	14	141	569.96
18	16	123	63.236
19	15	132	55.782
20	15	143	12.266
21	18	159	188.177
22	12	90	1012.078
23	12	112	204.769

Figure 13: Details of true errors found in the 26 DDK sample drivers. # functions means the number of C functions from the device driver or OS model that occur in the path; # steps indicates the number of assignments, conditional checks, function calls or function returns that occured in the error path.

This typically results in false errors, which lead us to refine the kernel model.

Second, the kernel APIs for plug-and-play and power management are quite complex, with many corner cases. The rules for these APIs were hard to get right. Often, kernel experts in these areas would disagree with one another about subtle points in the rules. As a result, we would develop a rule and have to iterate many times with the experts, showing them errors found by SDV and then refining the rules if the errors were false. This took a tremendous amount of time and energy.

Third, the harness only tries a limited set of execution sequences of dispatch routines (see Section 3). Suppose that a device driver programmer knows that a certain dispatch routine will always be called before others and sets up important invariants that the later dispatch routines depend on. If this sequence is not encoded in the harness, false errors may result. Unfortunately, it is too expensive to sequentially execute all possible interleavings of the dispatch routines.

7.2 Performance of SDV

We now present results about the performance of SDV on the 26 drivers from the DDK on the 64 rules supplied with SDV. (These runs were performed on a Pentium 3.06 GHz dual processor machine with hyper-threading and 2Gb of RAM.) There are a total of 1664 separate checks (checking a driver against a rule) performed. Of these checks, 885

Avg. total predicates in scope	8.012
Avg. global predicates	6.194
Avg. min. local predicates per function	0.938
Avg. max. local predicates per function	4.550

Figure 14: Averages regarding predicates generated during checking of 26 DDK sample drivers.

trivially pass because the `error` routine is not reachable in the call graph of the instrumented program (see Figure 5 and Section 5). The remaining checks break down into the following categories:

Pass	661
Error found	32
Abstraction failure	24
Tool failure	2
Timeout	64

So, we see that SDV is able to automatically prove that the majority of checks pass (1546/1664 = .93). Of the remaining 118 checks, SDV found 32 errors, failed in 26 cases and timed out in 64 cases with no definite result (a check times out after 2000 seconds = 33.33 minutes). The failure cases are broken into two categories: *Abstraction failures* are when the tool is unable to eliminate a known false error path. These account for 24 checks. *Tool failures* are undiagnosed errors in the SDV tool.

Of the 717 checks in which the model checking engine ran and completed with a definite result ("Pass" or "Error"), it iterated 5.63 times on average with a standard deviation of 11.21 iterations. That is, on average, five false errors (infeasible error path) were encountered and eliminated before the engine was able to complete a proof or find a feasible error path through the device driver. The median number of iterations is two, which deviates substantially from the average. That is, about half of the runs require only one iteration. The average run-time of the 717 checks is 101 seconds with a standard deviation of 267 seconds and a median run-time of 17 seconds. There is a substantial variation in run-time because the introduction of each new predicate by refinement doubles the potential state space to be explored. The average peak memory consumption of SDV over these checks was 30.5Mb. Generally, SDV runs effectively in under half a gigabyte of memory (though in certain situations it can become a memory hog if the binary decision diagram data structures used by the CHECK module blow-up in size).

In Section **??**, we postulated that API usage rules could be checked in device drivers by tracking a relatively small amount of state when compared to the entire state of the device driver. To demonstrate this we collected data about the number of predicates generated when checking the rules over the DDK device drivers. These statistics were collected only in the cases where the API usage rule was actually applicable to the driver. Figure 14 shows that, on average, eight predicates (Boolean variables) are needed at each program location during SDV's analysis. This is well within the scope of the model checking technology that SDV uses.

8. DISCUSSION

We now discuss some of the ways in which SDV could be extended to increase its applicability, precision and efficiency.

8.1 Memory Safety

When SDV says that a driver passes a rule, this is a guarantee that the tool is able to make after examining all the code paths. However the guarantee comes with some caveats. The soundness of SDV depends critically on the assumption that the device driver does not have wild pointers. That is, SDV does not check for the memory safety of a device driver, but assumes it. Another analysis (see for example CCURED [41]) is needed to discharge the assumption of memory-safe behavior.

8.2 Concurrency with Shared Memory

SDV currently analyzes each device driver in isolation with a sequential semantics, whereas in reality device drivers execute in the multi-threaded environment of the operating system. While many SDV's rules are motivated by concurrency issues, since SDV only analyzes one thread at a time it will miss errors that only are exhibited in the presence of more than one thread. That is, SDV does not detect errors that result from the interleavings of multiple threads. A new tool KISS [42] has been developed on top of SDV to find some classes of concurrency errors. KISS is not sound (i.e. if KISS reports that no race conditions have been found this does not guarentee that no race conditions exist). We are also investigating sound methods of supporting multiple threads in SDV through extensions to the CHECK module [20].

8.3 Integer and Bit-level Operations

SDV uses an automatic theorem prover [2] to implement symbolic simulation of C programs in its REFINE module. This theorem prover (and provers like it, such as SIMPLIFY [28]) treat numbers as unbounded integers, rather than fixed-width bitvectors. This can lead to cases where SDV reports both false errors and also misses errors (in cases where the error is due to overflow).

Furthermore, bit-level operations are treated as uninterpreted functions, which means that SDV may produce additional false errors. Because we have not included rules that rely on the values of bitwise operations, we find that few false errors can be attributed to our approximate modeling of bit vector operations, but this is an area for improvement. See [17, 18, 19] for recent work on this subject.

8.4 Liveness and Termination

It would be nice to show that a driver always makes progress. For example, we would like to show that when given an I/O request packet, a driver eventually will cancel or complete the packet. However, such a progress property is not a safety property but a *liveness* property. SDV currently does not support analysis of such properties. See [10, 21, 22] for recent work in this area.

8.5 Supporting Binary-level Analysis

SDV is a source-level tool, meaning that bugs introduced during compilation will be missed. In principle we could adapt SDV for use on binaries, perhaps using techniques described in [30].

8.6 Scaling SDV

There are a number of dimensions to scaling SDV. One is to get SDV to run efficiently on larger drivers. Work in the BLAST project has shown how to greatly increase the efficiency of the abstract-check-refine loop through the use of incremental analysis [35] and better predicate generation [34]. We also have found that SDV performs many refinements because its pointer analysis [25] is not field-sensitive. We believe that replacing SDV's pointer analysis with a field-sensitive one would greatly decrease the number of calls to the Refine module and increase SDV's efficiency.

Another challenge related to scaling is to develop API usage rules for other driver models. As we have mentioned above, KMDF is a new driver model that we have developed rules for. But there are many legacy driver models supported by Windows, including networking, storage, audio, display, etc. Each of these "models" is a library that abstracts away from the WDM driver model but has its own set of rules. It is an open question of how to scale rule development in the face of a large number of APIs.

9. CONCLUSION

Device drivers provide the mechanism by which any developer can add functionality into the Windows kernel. But kernel-level modules are hard to develop, and hard to test. SDV is an automatic tool that attempts to prove the correctness of these device drivers with respect to a set of kernel API usage rules. It has been used to find a number of deep and hard-to-reproduce errors within device drivers for the Windows operating system, including those that are distributed as a part of the Windows driver development kit.

Like a dataflow analyzer, SDV analyzes all code paths. However, it uses iterative refinement to greatly reduce false errors. This iterative refinement, also known as counterexample-guided abstraction refinement, distinguishes SDV from other dataflow analysis tools that do not perform refinement. Perhaps surprisingly, even though the analysis engine can produce false errors in theory, in practice we find that all the false errors were inaccuracies either in the rules or in the environment model, and we continually refine both the rules and the environment model whenever we notice such false errors.

Acknowledgements

We would like to thank Nar Ganapathy, Bob Rinne, Rob Short, Jim Larus, Adrian Oney, Amitabh Srivastava, and Peter Wieland for their support of the SDV project.

10. REFERENCES

[1] T. Ball, B. Cook, S. Das, and S. K. Rajamani. Refining approximations in software predicate abstraction. In *TACAS 04: Tools and Algorithms for the Construction and Analysis of Systems*, pages 388–403, 2004.

[2] T. Ball, B. Cook, S. K. Lahiri, and L. Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In *CAV 04: Computer-Aided Verification*, pages 457–461, 2004.

[3] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 01: Programming Language Design and Implementation*, pages 203–213, 2001.

[4] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *POPL 03: Principles of programming languages*, pages 97–105, 2003.

[5] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstractions for model checking C programs. In *TACAS 01: Tools and Algorithms for Construction and Analysis of Systems*, pages 268–283, 2001.

[6] T. Ball, A. Podelski, and S. K. Rajamani. On the relative completeness of abstraction refinement. In *TACAS 02: Tools and Algorithms for Construction and Analysis of Systems*, pages 158–172, April 2002.

[7] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *SPIN 00: SPIN Workshop*, pages 113–130, 2000.

[8] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 01: SPIN Workshop*, pages 103–122, 2001.

[9] T. Ball and S. K. Rajamani. Bebop: A path-sensitive interprocedural dataflow engine. In *PASTE 01: Workshop on Program Analysis for Software Tools and Engineering*, pages 97–103, 2001.

[10] A. Bradley, Z. Manna, and H. Sipma. Linear ranking with reachability. In *CAV 05: Computer-Aided Verification*, pages 491–504, 2005.

[11] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[12] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software-Practice and Experience*, 30(7):775–802, June 2000.

[13] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in c. In *ICSE 03: International Conference on Software Engineering*, pages 385–395, 2003.

[14] H. Chen, D. Dean, and D. Wagner. Model checking one million lines of c code. In *NDSS 04: Network and Distributed System Security Symposium*, 2004.

[15] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *SOSP 01: Symposium on Operating System Principles*, pages 73–88, 2001.

[16] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00: Computer Aided Verification*, pages 154–169, 2000.

[17] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI–C programs using SAT. *Formal Methods in System Design (FMSD)*, 25:105–127, September–November 2004.

[18] B. Cook and G. Gonthier. Using Stalmårck's algorithm to prove inequalities. In *ICFEM 05: Conference on Formal Engineering Methods*, pages 330–344, 2005.

[19] B. Cook, D. Kroening, and N. Sharygina. Cogent: Accurate theorem proving for program verification. In *CAV 05: Computer-Aided Verification*, pages 296–300, 2005.

[20] B. Cook, D. Kroening, and N. Sharygina. Symbolic model checking for asynchronous boolean programs.

In *SPIN 01: SPIN Workshop*, pages 75–90, 2005.

[21] B. Cook, A. Podelski, and A. Rybalchenko. Abstraction refinement for termination. In *SAS 05: Static Analysis Symposium*, pages 87–101, 2005.

[22] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI 06: Programming Language Design and Implementation*, 2006.

[23] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *POPL 77: Principles of Programming Languages*, pages 238–252, 1977.

[24] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTREÉ analyzer. In *ESOP 05: European Symposium on Programming*, pages 21–30, 2005.

[25] M. Das. Unification-based pointer analysis with directional assignments. In *PLDI 00: Programming Language Design and Implementation*, pages 35–46, 2000.

[26] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI 02: Programming Language Design and Implementation*, pages 57–68, June 2002.

[27] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI 01: Programming Language Design and Implementation*, pages 59–69, 2001.

[28] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, 2003.

[29] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI 00: Operating System Design and Implementation*, pages 1–16. Usenix Association, 2000.

[30] G. Balakrishnan et al. Model checking x86 executables with CodeSurfer/x86 and WPDS++. In *CAV 05: Computer-Aided Verification*, 2005.

[31] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI 02: Programming Language Design and Implementation"*, pages 234–245, 2002.

[32] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williams. Safe hardware access with the Xen virtual machine monitor. In *OASIS'04: Workshop on Operating System and Architectural Support for the on demand IT InfraStructure*, June 2004.

[33] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *PLDI 02: Programming Language Design and Implementation*, pages 69–82, 2002.

[34] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL 04: Principles of Programming Languages*, pages 232–244, 2004.

[35] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL 02: Principles of Programming Languages*, pages 58–70, January 2002.

[36] R.P. Kurshan. *Computer-aided Verification of Coordinating Processes*. Princeton University Press, 1994.

[37] S. Lahiri, T. Ball, and B. Cook. Predicate abstraction via symbolic decision procedures. In *CAV 05: Computer-Aided Verification*, pages 24–38, 2005.

[38] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.

[39] J. R. Larus, T. Ball, M. Das, Rob DeLine, M. Fähndrich, J. Pincus, S. K. Rajamani, and R. Venkatapathy. Righting software. *IEEE Software*, 21(3):92–100, May/June 2004.

[40] K. R. M. Leino and G. Nelson. An extended static checker for Modula-3. In *CC 98: Compiler Construction*, pages 302–305, 1998.

[41] G. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *POPL 02: Principles of Programming Languages*, pages 128–139, January 2002.

[42] S. Qadeer and D. Wu. KISS: keep it simple and sequential. In *PLDI 04: Programming Language Design and Implementation*, pages 14–24, 2004.

[43] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[44] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *SOSP 03: Symposium on Operating System Principles*, pages 207–222, June 2003.

[45] M. Y. Vardi and P. Wolper. An automata theoretic apporach to automatic program verification. In *LICS 86: Logic in Computer Science*, pages 332–344. IEEE Computer Society Press, 1996.