# Analyzing Stripped Device-Driver Executables[⋆]

Gogul Balakrishnan[⋆⋆,1] and Thomas Reps[2,3]

[1] NEC Laboratories America, Inc.
[2] University of Wisconsin
[3] GrammaTech, Inc.
`bgogul@nec-labs.com`, `reps@cs.wisc.edu`

**Abstract.** This paper sketches the design and implementation of Device-Driver Analyzer for x86 (DDA/x86), a prototype analysis tool for finding bugs in stripped Windows device-driver executables (i.e., when neither source code nor symbol-table/debugging information is available), and presents a case study. DDA/x86 was able to find known bugs (previously discovered by source-code-based analysis tools) along with useful error traces, while having a reasonably low false-positive rate.

This work represents the first known application of automatic program verification/analysis to stripped industrial executables, and allows one to check that an executable does not violate known API usage rules (rather than simply trusting that the implementation is correct).

## 1 Introduction

A device driver is a program in the operating system that is responsible for managing a hardware device attached to the system. In Windows, a (kernel-level) device driver resides in the address space of the kernel, and runs at a high privilege level; therefore, a bug in a device driver can cause the entire system to crash. The Windows kernel API [27] requires a programmer to follow a complex set of rules: (1) a call to the functions *IoCallDriver* or *PoCallDriver* must occur only at a certain interrupt request level, (2) the function *IoCompleteRequest* should not be called twice with the same parameter, etc.

The device drivers running in a given Windows installation are one of the sources of instability in the Windows platforms: according to Swift et al. [31], bugs in kernel-level device drivers cause 85% of the system crashes in Windows XP. Because of the complex nature of the Windows kernel API, the probability of introducing a bug when writing a device driver is high. Moreover, drivers are typically written by less-experienced or less-skilled programmers than those who wrote the Windows kernel itself.

Several approaches to improve the reliability of device drivers have been previously proposed [10, 12, 15, 31]. Swift et al. [30, 31] propose a runtime approach that works on executables; they isolate the device driver in a light-weight protection domain to reduce the possibility of whole-system crashes. Because their

[⋆] Supported by NSF under grants CCF-0540955 and CCF-0524051.
[⋆⋆] Work performed while at the University of Wisconsin.

method is applied at runtime, it may not prevent all bugs from causing whole-system crashes. Other approaches [10–12, 22] are based on static program analysis of a device driver's source code. Ball et al. [10, 12] developed the Static Driver Verifier (SDV), a tool based on model checking to find bugs in device-driver source code. A kernel API usage rule is described as a finite-state machine (FSM), and SDV analyzes the source code for the driver to determine whether there is a path in the driver that violates the rule.

Our work, which is incorporated in a prototype tool called Device-Driver Analyzer for x86 (DDA/x86), is also based on static analysis, but in contrast to the work cited above, DDA/x86 checks properties of stripped Windows device-driver executables; i.e., neither source code nor symbol-table/debugging information need be available (although DDA/x86 can use debugging information, such as Windows .pdb files, if it is available). Thus, DDA/x86 can provide information that is useful in the common situation where one needs to install a device driver for which source code has not been furnished.

Microsoft has a program for signing Windows device drivers, called Windows Hardware Quality Lab (WHQL) testing. Device vendors submit driver executables to WHQL, which runs tests on different hardware platforms with different versions of Windows, reviews the results, and, if the driver passes the tests, creates a digitally signed certificate for use during installation that attests that Microsoft has performed some degree of testing. However, there is anecdotal evidence that device vendors have tried to cheat [2]. A tool like DDA/x86 could allow static analysis to play a role in such a certification process.

Even if you have a driver's source code (and can build an executable) and also have tools for examining executables equipped with symbol-table/debugging information, this would not address the effects of the optimizer. If you want to look for bugs in an optimized version, you would have a kind of "partially stripped" executable, due to the loss of debugging information caused by optimization. This is a situation where our techniques for analyzing stripped executables should be of assistance.

A skeptic might question how well an analysis technique can perform on a stripped executable. §4 presents some quantitative results about how well the answers obtained by DDA/x86 compare to those obtained by SDV; here we will just give one example that illustrates the ability of DDA/x86 to provide information that is qualitatively comparable to the information obtained by SDV. Fig. 1 shows fragments of the witness traces reported by SDV (Fig. 1(a)) and DDA/x86 (Fig. 1(b)) for one of the examples in our test suite. Fig. 1 shows that in this case the tools report comparable information: the three shaded areas in Fig. 1(b) correspond to those in Fig. 1(a).

Although not illustrated by Fig. 1, there are ways in which DDA/x86 can provide higher-fidelity answers than tools based on analyzing source code. This may seem counterintuitive, but the reason is that DDA/x86 works at a level in which many platform-specific features are revealed, such as memory-layout details (e.g., the offsets of variables in activation records and padding between fields of a `struct`). Because the compiler is in charge of such choices, and may

```
KeInitializeEvent(&event, NotificationEvent, FALSE);

IoSetCompletionRoutine(Irp,
                        (PIO_COMPLETION_ROUTINE) MouFilter_Complete,
                        &event,
                        TRUE,
                        TRUE,
                        TRUE); // No need for Cancel

status = IoCallDriver(devExt->TopOfStack, Irp);

if (STATUS_PENDING == status) {
    KeWaitForSingleObject(
        &event,
        Executive, // Waiting for reason of a driver
        KernelMode, // Waiting in kernel mode
        FALSE, // No alert
        NULL); // No timeout
}

if (NT_SUCCESS(status) && NT_SUCCESS(Irp->IoStatus.Status)) {
    //
    // As we are successfully now back from our start device
    // we can do work.
    //
    devExt->Started = TRUE;
    devExt->Removed = FALSE;
    devExt->SurpriseRemoved = FALSE;
}

//
// We must now complete the IRP, since we stopped it in the
// completion routine with MORE_PROCESSING_REQUIRED.
//
Irp->IoStatus.Status = status;
Irp->IoStatus.Information = 0;
IoCompleteRequest(Irp, IO_NO_INCREMENT);

break;
```

(a)

```
    push 1
    push 1
    push 1
    lea ecx, dword ptr [ebp + var_1C]
    push ecx
    push sub_1002270
    mov edx, dword ptr [ebp + arg_4]
    push edx
    call dword ptr [_sdv_IoSetCompletionRoutine@24]
    mov edx, dword ptr [ebp + arg_4]
    mov eax, dword ptr [ebp + var_4]
    mov ecx, dword ptr [eax + 8]
    call dword ptr [@IofCallDriver@8]
    mov dword ptr [ebp + var_20], eax
    cmp dword ptr [ebp + var_20], 103h
    jnz loc_10013E1
    push 0
    push 0
    push 0
    push 0
    lea ecx, dword ptr [ebp + var_1C]
    push ecx,
    call dword ptr [_sdv_KeWaitForSingleObject@20]

loc_10013E1:
    cmp dword ptr [ebp + var_20], 0
    jl loc_1001405
    mov edx, dword ptr [ebp + arg_4]
    cmp dword ptr [edx + 18h], 0
    jl loc_1001405
    mov eax, dword ptr [ebp + var_4]
    mov byte ptr [eax + 30h], 1
    mov ecx, dword ptr [ebp + var_4]
    mov byte ptr [ecx + 32h], 0
    mov edx, dword ptr [ebp + var_4]
    mov byte ptr [edx + 31h], 0

loc_1001405:
    mov eax, dword ptr [ebp + arg_4]
    mov ecx, dword ptr [ebp + var_20]
    mov dword ptr [eax + 18h], ecx
    mov edx, dword ptr [ebp + arg_4]
    mov dword ptr [edx + 1Ch], 0
    push 0
    mov eax, dword ptr [ebp + arg_4]
    push eax
    call dword ptr [_sdv_IoCompleteRequest@8]
    jmp loc_10014B0
```

(b)

**Fig. 1.** (a) SDV trace; (b) DDA/x86 trace. The three shaded areas in (b) correspond to those in (a).

also restructure the computation in certain ways, the machine-code level at which DDA/x86 works is closer than the source-code level to what is actually executed.

Elsewhere [4, 8], we have called this the WYSINWYX phenomenon (**W**hat **Y**ou **S**ee **I**s **N**ot **W**hat **Y**ou e**X**ecute): computers execute the instructions of programs that have been complied, and not the source code itself; compilation effects can be important if one is interested in better diagnosis of the causes of bugs, or in detecting security vulnerabilities. A Microsoft report on writing kernel-mode drivers in C++ recommends examining "... the object code to be sure it matches your expectations, or at least will work correctly in the kernel environment" [3]. As discussed in §4, we encountered a few cases of the WYSINWYX phenomenon in our experiments, although these concerned the hand-written environment harnesses that we picked up from SDV [10, 12].

This paper describes the design and implementation of DDA/x86, and presents a case study in which we used it to find problems in Windows device drivers by analyzing the drivers' stripped executables. The key idea that allows DDA/x86 to achieve a substantial measure of success was to combine the algorithm for memory-access analysis [4–6] from CodeSurfer/x86 [7] with the path-sensitive method of interpreting property automata from ESP [17]. The resulting algorithm explores an over-approximation of the set of reachable states, and hence can verify correctness by determining that all error configurations are unreachable. The contributions of the work include

– DDA/x86 can analyze *stripped device-driver executables*, and thus provides a capability not found in previous tools for analyzing device drivers [11, 22].
– Our case study shows that this approach is viable. DDA/x86 was able to identify some known bugs (previously discovered by source-code-based analysis tools) along with useful error traces, while having a reasonably low false-positive rate: On a corpus of 17 device-driver executables, 10 were found to pass the *PendedCompletedRequest* rule (definitely no bug), 5 false positives were reported, and 2 were found to have real bugs—for which DDA/x86 supplied feasible error traces.
– We developed a novel, low-cost mechanism for instrumenting a dataflow-analysis algorithm to provide witness traces.

One of the challenges that we faced was to find ways of coping with the differences that arise when property checking is performed at the machine-instruction level, rather than on an IR created from source code. In particular, the domains of discourse—the alphabets of actions to which the automata respond—are different in the two situations. This issue is discussed in §4.

The remainder of the paper is organized as follows: §2 provides background on recovering intermediate representations (IRs) from executables. §3 describes the extensions that we made to our algorithm for IR-recovery from low-level code to perform path-sensitive property checking. §4 presents experimental results. Related work is discussed in §5.

## 2 Background on Intermediate-Representation Recovery

DDA/x86 makes use of the IR-recovery algorithms of CodeSurfer/x86 [4–7]. This section explains some of the ideas used in those algorithms that are important to understanding how they were extended to support path-sensitivity.

The IR-recovery algorithms of CodeSurfer/x86 recover from a device-driver executable IRs that are similar to those that would be available had one started from source code. CodeSurfer/x86 recovers IRs that represent control-flow graphs (CFGs), with indirect jumps resolved; a call graph, with indirect calls resolved; information about the program's variables; possible values of pointer variables; sets of used, killed, and possibly-killed variables for each CFG node; and data dependences. The techniques employed by CodeSurfer/x86 do not rely on debugging information being present, but can use available debugging information (e.g., Windows .pdb files) if directed to do so.

The analyses used in CodeSurfer/x86 (see [4–6]) are a great deal more ambitious than even relatively sophisticated disassemblers, such as IDAPro [24]. At the technical level, they address the following problem: *Given a (possibly stripped) executable E, identify the procedures, data objects, types, and libraries* *that it uses, and, for each instruction $I$ in $E$ and its libraries, for each interprocedural calling context of $I$, and for each machine register and variable $V$ in scope at $I$, statically compute an accurate over-approximation to the set of values that $V$ may contain when $I$ executes.*

**Variable and Type Discovery.** One of the major stumbling blocks in analyzing executables is the difficulty of recovering information about variables and types, especially for aggregates (i.e., structures and arrays). When performing source-code analysis, the programmer-defined variables provide us with the compartments for tracking data manipulations. When debugging information is absent, an executable's data objects are not easily identifiable. Consider, for instance, an access on a source-code variable `x` in some source-code statement. At the machine-instruction level, an access on `x` is performed either directly—by specifying an absolute address—or indirectly—through an address expression of the form "[*base* + *index* × *scale* + *offset*]", where *base* and *index* are registers and *scale* and *offset* are integer constants. The variable and type-discovery phase of CodeSurfer/x86 [4, 6] recovers information about variables that are allocated globally, locally (i.e., on the run-time stack), and dynamically (i.e., from the heap). The recovered variables, called *a-locs* (for "abstract locations") are the basic variables used in the extension of the VSA algorithm described in §3.

To accomplish this task, CodeSurfer/x86 makes use of a number of analyses, and the sequence of analyses performed is itself iterated [4, 6]. On each round, CodeSurfer/x86 uses VSA to identify an over-approximation of the memory accesses performed at each instruction. Subsequently, the results of VSA are used to perform aggregate structure identification (ASI) [28], which identifies commonalities among accesses to different parts of an aggregate data value, to refine the current set of a-locs. The new set of a-locs are used to perform another round of VSA. If the over-approximation of memory accesses computed by VSA improves from the previous round, the a-locs computed by the subsequent round of ASI may also improve. This process is repeated as long as desired, or until the process converges. By this means, CodeSurfer/x86 bootstraps its way to a set of a-locs that serve as proxies for the program's original variables.

## 3 Property Checking in Executables using VSA

This section describes the extensions that we made to our IR-recovery algorithm to perform path-sensitive property checking. Consider the following memory-safety property: `p` should not be dereferenced if its value is `NULL`. Fig. 2 shows an FSM that checks for property violations. One approach to determining if there is a null-pointer dereference in the executable is to start from the initial state (UNSAFE) at the entry point of the executable, and find an over-approximation of the set of reachable states at each statement in the executable. This can be done by determining the states for the successors at each statement based on the transitions in the FSM that encodes the memory-safety property.

Another approach is to use abstract interpretation to determine the abstract memory configurations at each statement in the routine, and use the results to check the memory-safety property. For executables, we could use the information computed by the IR-recovery algorithms of CodeSurfer/x86 [7]. For instance, for each instruction $I$ in an executable, the value-set analysis (VSA) algorithm [4–6] used in CodeSurfer/x86 determines an over-approximation of the set of memory addresses and numeric values held in each register and variable when $I$ executes.

Suppose that we have the results of VSA and want to use them to check the memory-safety property; the property can be checked as follows:

*If the abstract set of addresses and numeric values computed for* `p` *possibly contains* `NULL` *just before a statement, and the statement dereferences* `p`*, then the memory-safety property is potentially violated.*

Unfortunately, the approaches described above would result in a lot of false positives because they are not path-sensitive. To overcome the limitations of the two approaches described above, DDA/x86 follows Das et al. [17] and Fischer et al. [20], who showed how to obtain a degree of path-sensitivity by combining the propagation of automaton states with the propagation of abstract-state values during abstract interpretation. The remainder of this section describes how the propagation of property-automaton states can be incorporated into the VSA algorithm to obtain a degree of path-sensitivity.

To simplify the discussion, the ideas are initially described for a simplified version of VSA, called *context-insensitive* VSA [5]; the combination of automaton-state propagation with the context-sensitive version of VSA [4, Ch. 3] is discussed at the end of the section. The context-insensitive VSA algorithm associates each program point with an $\mathsf{AbsEnv}$ value [4, 6], which represents a set of concrete (i.e., run-time) states of a program. An element in the $\mathsf{AbsEnv}$ domain associates each a-loc and register in the executable with an abstract value that represents a set of memory addresses and numeric values. Let $\mathsf{State}$ denote the set of property-automaton states. The path-sensitive VSA algorithm associates each program point with an $\mathsf{AbsMemConfig}^{\mathrm{ps}}$ value, where $\mathsf{AbsMemConfig}^{\mathrm{ps}} = ((\mathsf{State} \times \mathsf{State}) \to \mathsf{AbsEnv}_{\perp})$.

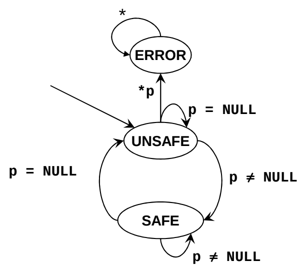

**Fig. 2.** An FSM that encodes the rule that pointer p should not be dereferenced if it is `NULL`.

In the pair of property-automaton states at a node $n$, the first component refers to the state of the property automaton at the enter node of the procedure to which node $n$ belongs, and the second component refers to the current state of the property automaton at node $n$. If an $\mathsf{AbsEnv}$ entry for the pair $\langle s_0, s_{cur}\rangle$ exists at node $n$, then $n$ is possibly reachable with the property automaton in state $s_{cur}$ from a memory configuration at the enter node of the procedure in which the property automaton was in state $s_0$.

The path-sensitive VSA algorithm is shown in Fig. 3. The worklist consists of triples of the form $\langle \mathsf{State}, \mathsf{State}, \mathsf{Node}\rangle$. A triple $\langle \mathit{enter\_state}, \mathit{cur\_state}, n\rangle$ is selected from the worklist, and for each successor edge of node $n$, a new $\mathsf{AbsEnv}$ value is computed by applying the corresponding abstract transformer (line [11]).

After computing a new AbsEnv value, the set of pairs of states for the successor is identified (see the *GetSuccStates* procedure in Fig. 3). For an intraprocedural edge *pred*$\rightarrow$*succ*, the set of pairs of states for the target of the edge is obtained by applying the *NextStates* function to $\langle$*enter_state*, *cur_state*$\rangle$ (line [34]). The *NextStates* function pairs *enter_state* with all possible second-component states according to the property automaton's transition relation for edge *pred*$\rightarrow$*succ*. For a call$\rightarrow$enter edge, the only new state pair is the pair $\langle$*cur_state*, *cur_state*$\rangle$ (line [30]). For an exit$\rightarrow$end-call edge, the set of pairs of states for the end-call node is determined by examining the set of pairs of states at the corresponding call (lines [24]–[28]); for each $\langle$*call_enter_state*, *call_cur_state*$\rangle$ at the call node such that (*call_cur_state* = *enter_state*), the pair $\langle$*call_enter_state*, *cur_state*$\rangle$ is added to the result.

Note that the condition (*call_cur_state* = *enter_state*) at line [25] checks if $\langle$*enter_state*, *cur_state*$\rangle$ at the exit node is reachable, according to the property automaton, from $\langle$*call_enter_state*, *call_cur_state*$\rangle$ at the call node. The need to check the condition (*call_cur_state* = *enter_state*) at an exit node is the reason for maintaining a pair of states at each node. If we do not maintain a pair of states, it would not be possible to determine the property-automaton states at the call that reach the given property-automaton state at the exit node. (In essence, we are doing a natural join a tuple at a time: the subset of State $\times$ State at the call node represents a reachability relation $R_1$ for the property automaton's possible net change in state as control moves from the caller's enter node to the call site; the subset of State $\times$ State at the exit node represents a reachability relation $R_2$ for the automaton's net change in state as control moves from the callee's enter node to the exit node. The subset of State $\times$ State at the end-call node, representing a reachability relation $R_3$, is their natural join, given by $R_3(x, y) = \exists z.\ R_1(x, z) \wedge R_2(z, y)$. Thus, technically our extension amounts to the use of the reduced cardinal power [16, 17, 20] of the edges in the transitive closure of the automation's transition relation and the original VSA domain.)

Finally, in the AbsMemConfig$^{\text{ps}}$ value for the successor node, the AbsEnv values for all the pairs of states that were identified by *GetSuccStates* are updated with the newly computed AbsEnv value (see the *Propagate* function in Fig. 3).

It is trivial to combine the path-sensitive VSA algorithm in Fig. 3 and the context-sensitive VSA algorithm to get a VSA algorithm that can distinguish paths as well as calling contexts to a limited degree. In the combined algorithm, each node is associated with a value from the following domain (where CallString$_k$ represents the set of call-string suffixes of length up to $k$ [29]):

$$\mathsf{AbsMemConfig}^{\text{ps-cs}} = ((\mathsf{CallString_k} \times \mathsf{State} \times \mathsf{State}) \rightarrow \mathsf{AbsEnv}_\perp).$$

## 4 Experiments

This section presents a case study in which we used DDA/x86 to analyze the executables of Windows device drivers. The study was designed to test how well different extensions of the VSA algorithm could detect problems in Windows device drivers by analyzing device-driver executables—without accessing source

```
decl $worklist$: $set$ of $\langle \mathsf{State}, \mathsf{State}, \mathsf{Node} \rangle$

proc $PathSensitiveVSA()$
    $worklist := \{\langle \mathtt{StartState}, \mathtt{StartState}, \mathtt{enter} \rangle\}$
    $\text{absMemConfig}^{\text{ps}}_{\mathtt{enter}}[\langle \mathtt{StartState}, \mathtt{StartState} \rangle]$ := Initial values of global a-locs and esp
    while ($worklist \neq \emptyset$) do
        Select and remove a triple $\langle enter\_state, cur\_state, n \rangle$ from $worklist$
        $m$ := Number of successors of node $n$
        for $i = 1$ to $m$ do
            $succ := GetSuccessor(n, i)$
            $edge\_amc := AbstractTransformer(n \rightarrow succ, \text{absMemConfig}^{\text{ps}}_{n}[\langle enter\_state, cur\_state \rangle])$
            $\underline{succ\_states := GetSuccStates(enter\_state,\ cur\_state,\ n,\ succ)}$
            for (each $\langle succ\_enter\_state, succ\_cur\_state \rangle \in succ\_states$) do
                $Propagate(enter\_state,\ succ\_enter\_state,\ succ\_cur\_state,\ succ,\ edge\_amc)$
            end for
        end for
    end while
end proc

proc $GetSuccStates(enter\_state$: State, $cur\_state$: State, $pred$: Node, $succ$: Node): $set$ of
  $\langle \mathsf{State}, \mathsf{State} \rangle$
    $result := \emptyset$
    if ($pred$ is an exit node and $succ$ is an end-call node) then
        Let $c$ be the call node associated with $succ$
        for $\underline{\text{each } \langle call\_enter\_state, call\_cur\_state \rangle \text{ in absMemConfig}^{\text{ps}}_{c}}$ do
            if $\underline{(call\_cur\_state = enter\_state)}$ then
                $\underline{result := result \cup \{\langle call\_enter\_state, cur\_state \rangle\}}$
            end if
        end for
    else if ($pred$ is a call node and $succ$ is an enter node) then
        $\underline{result := \{\langle cur\_state, cur\_state \rangle\}}$
    else
        // Pair $enter\_state$ with all possible second-component states according to
        // the property automaton's transition relation for input edge $pred \rightarrow succ$
        $\underline{result := NextStates(pred \rightarrow succ,\ \langle enter\_state, cur\_state \rangle)}$
    end if
    return $result$
end proc

proc $Propagate(pred\_enter\_state$: State, $enter\_state$: State, $cur\_state$: State, $n$: Node, $edge\_amc$:
  AbsEnv)
    $old := \text{absMemConfig}^{\text{ps}}_{n}[\langle enter\_state, cur\_state \rangle]$
    if $n$ is an end-call node then
        Let $c$ be the call node associated with $n$
        $edge\_amc := MergeAtEndCall(edge\_amc, \text{absMemConfig}^{\text{ps}}_{c}[\langle enter\_state, pred\_enter\_state \rangle])$
    end if
    $new := old \sqcup^{\text{ae}} edge\_amc$
    if ($old \neq new$) then
        $\text{absMemConfig}^{\text{ps}}_{n}[\langle enter\_state, cur\_state \rangle] := new$
        $worklist := worklist \cup \{\langle enter\_state, cur\_state, n \rangle\}$
    end if
end proc
```

**Fig. 3.** Path-sensitive VSA algorithm. (The function *MergeAtEndCall* merges information from the abstract state at an exit node with information from the abstract state at the call node (cf. [25]). Underlining indicates an action that manages or propagates property-state information.)

code, symbol-tables, or debugging information. In particular, if DDA/x86 were successful at finding the bugs that the Static Driver Verifier (SDV) [10, 12] tool finds in Windows device drivers, that would be powerful evidence that our approach is viable—i.e., that it will be possible to find previously undiscovered bugs in device drivers for which source code is not available, or for which compiler/optimizer effects make source-code analysis unsafe. We selected a subset of drivers from the Windows Driver Development Kit (DDK) [1] release 3790.1830 for the case study. For each driver, we obtained an executable by compiling the driver source code along with the harness and the OS environment model [10] of the SDV toolkit. (Thus, as in SDV and other source-code-analysis tools, the harness and OS environment models are analyzed; however, DDA/x86 analyzes the executable code that the compiler produces for the harness and the models. This creates certain difficulties, which are discussed below.)

A device driver is analogous to a library that exports a collection of subroutines. Each subroutine exported by a driver implements an action that needs to be performed when the OS makes an I/O request (on behalf of a user application or when a hardware-related event occurs). For instance, when a new device is attached to the system, the OS invokes the `AddDevice` routine provided by the device driver; when new data arrives on a network interface, the OS calls the `DeviceRead` routine provided by the driver; etc. For every I/O request, the OS creates a structure called the "I/O Request Packet (IRP)", which contains such information as the type of the I/O request, the parameters associated with the request, etc.; the OS then invokes the appropriate driver's dispatch routine. The dispatch routine performs the necessary actions, and returns a value that indicates the status of the request. For instance, if a driver successfully completes the I/O request, the driver's dispatch routine calls the *IoCompleteRequest* API function to notify the OS that the request has been completed, and returns the value `STATUS_SUCCESS`. Similarly, if the I/O request is not completed within the dispatch routine, the driver calls the *IoMarkPending* API function and returns `STATUS_PENDING`.

A harness in the SDV toolkit is C code that simulates the possible calls to the driver that could be made by the OS. An application generates requests, which the OS passes on to the device driver. Both levels are modeled by the harness. The harness defined in the SDV toolkit acts as a client that exercises all possible combinations of the dispatch routines that can occur in two successive calls to the driver. The harness that was used in our experiments calls the following driver routines (in the order given below):

1. `DriverEntry`: initializes the driver's data structures and the global state.
2. `AddDevice`: simulates the addition of a device to the system.
3. The plug-and-play dispatch routine (called with an `IRP_MN_START_DEVICE` I/O request packet): simulates the starting of the device by the OS.
4. Some dispatch routine, deferred procedure call, interrupt service routine, etc.: simulates various actions on the device.
5. The plug-and-play dispatch routine (called with an `IRP_MN_REMOVE_DEVICE` I/O request packet): simulates the removal of the device by the OS.

6. `Unload`: simulates the unloading of the driver by the OS.

The OS environment model in the SDV toolkit consists of a collection of functions (written in C) that conservatively model the API functions in the Windows DDK. The models are conservative in the sense that they simulate all possible behaviors of an API function. For instance, if an API function `Foo` returns the value 0 or 1 depending upon the input arguments, the model for `Foo` consists of a non-deterministic `if` statement that returns 0 in the true branch and 1 in the false branch. Modeling the API functions conservatively enables a static-analysis tool to explore all possible behaviors of the API.

**WYSINWYX.** We had to make some changes to the OS models used in the SDV toolkit because SDV's models were never meant to be compiled and used, in compiled form, as models of the OS environment by an analyzer that works on machine instructions, such as DDA/x86. These problems showed up as instances of the WYSINWYX phenomenon. For instance, each driver has a device-extension structure that is used to maintain extended information about the state of each device managed by the driver. The number of fields and the type of each field in the device-extension structure is specific to a driver. However, in SDV's OS model, a single integer variable is used to represent the device-extension object. Therefore, in a driver executable built using SDV's models, when the driver writes to a field at offset $o$ of the device-extension structure, it would appear as a write to the memory address that is offset $o$ bytes from the memory address of the integer that represents the device-extension object.

We also encountered the WYSINWYX phenomenon while using SDV's OS models. For instance, the OS model uses a function named `SdvMakeChoice` to represent non-deterministic choice. However, the body of `SdvMakeChoice` only contains a single "`return 0`" statement.[4] Consequently, instead of exploring all possible behaviors of an API function, DDA/x86 would explore only a subset of the behaviors of the API function. We had to modify SDV's OS environment model to avoid such problems.

**Case Study.** We chose the following "*PendedCompletedRequest*" rule for our case study:

> *A driver's dispatch routine should not return `STATUS_PENDING` on an I/O Request Packet (IRP) if it has called IoCompleteRequest on the IRP, unless it has also called IoMarkIrpPending.*

Fig. 4 shows the FSM for this rule.[5]

We used the three different variants of the VSA algorithm listed in Tab. 1 for our experiments on a 64-bit Xeon 3GHz processor with 16GB (only 4GB/process) of memory, and Tab. 2 presents the results. The column labeled "Result" indicates whether the VSA algorithm reported that there is some node $n$ at which the `ERROR` state in the *PendedCompletedRequest* FSM is reachable, when one starts from the initial memory configuration at the entry node of the executable.

[4] According to T. Ball [9], the C front end used by SDV treats `SdvMakeChoice` specially.

[5] According to the Windows DDK documentation, *IoMarkPending* has to be called before *IoCompleteRequest*; however, the FSM defined for the rule in SDV is the one shown in Fig. 4. We used the same FSM for our experiments.

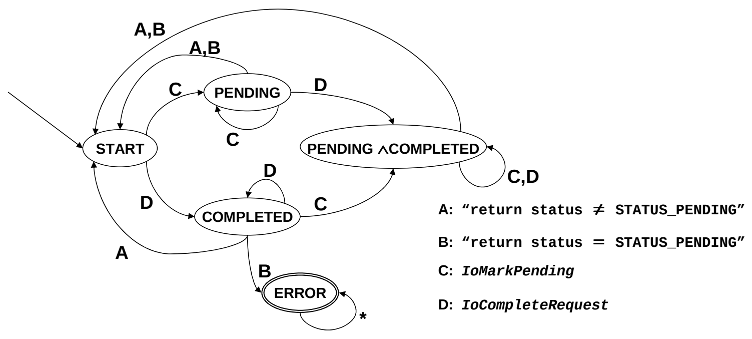

**Fig. 4.** Finite-state machine for the rule *PendedCompletedRequest*.

| Config. | A-locs | Property Automaton |
|---|---|---|
| ⊖ | IDAPro-based algorithm | Fig. 4 |
| ⊚ | ASI-based algorithm | Fig. 4 |
| ★ | ASI-based algorithm | Cross-product of the automata in Figs. 4 and 6 |

**Table 1.** Variants of the VSA algorithm used in the experiments.

Configuration '⊖' uses an algorithm that is similar to the one used in IDAPro to recover variable-like entities. That algorithm does not provide variables of the right granularity and expressiveness, and therefore, not surprisingly, configuration '⊖' reports many false positives for all of the drivers.[6]

Configuration '⊚', which uses only the *PendedCompletedRequest* FSM, also reports a lot of false positives. Fig. 5 shows an example that illustrates one of the reasons for the false positives in configuration '⊚'. As shown in the right column of Fig. 5, the set of values for `status` at the return statement (`P3`) for the property-automaton state `COMPLETED` contains both `STATUS_PENDING` and `STATUS_SUCCESS`. Therefore, VSA reports that the dispatch routine possibly violates the *PendedCompletedRequest* rule. The problem is as follows: because the state of the *PendedCompletedRequest* automaton is the same after both branches of the `if` statement at `P1` are analyzed, VSA merges the information from both of the branches, and therefore the correlation between `c` and `status` is lost after the statement at `P2`.

Fig. 6 shows an FSM that enables VSA to maintain the correlation between `c` and `status`. Basically, the FSM changes the abstraction in use, and enables VSA to distinguish paths in the executable based on the contents of the variable `status`. We refer to a variable (such as `status` in Fig. 6) that is used to keep track of the current status of the I/O request in a dispatch routine as the *status-variable*. To be able to use the FSM in Fig. 6 for analyzing an executable, it is

[6] In this case, a false positive reports that the `ERROR` state is (possibly) reachable at some node $n$, when, in fact, it is never reachable. This is sound (for the reachability question), but imprecise.

| | | | $\ominus$ | | $\circledcirc$ | | $\bigstar$ | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Driver | Procedures | Instructions | Result | Feasible Trace? | Result | Feasible Trace? | Result | Feasible Trace? | Time | Rounds |
| src/vdd/dosioctl/krnldrvr | 70 | 2824 | FP | - | √ | - | √ | - | 14s | 2 |
| src/general/ioctl/sys | 76 | 3504 | FP | - | √ | - | √ | - | 13s | 2 |
| src/general/tracedrv/tracedrv | 84 | 3719 | FP | - | √ | - | √ | - | 16s | 2 |
| src/general/cancel/startio | 96 | 3861 | FP | - | √ | - | √ | - | 12s | 2 |
| src/general/cancel/sys | 102 | 4045 | FP | - | √ | - | √ | - | 10s | 2 |
| src/input/moufiltr | 93 | 4175 | × | No | × | No | × | Yes | 3m 3s | 5 |
| src/general/event/sys | 99 | 4215 | FP | - | √ | - | √ | - | 20s | 2 |
| src/input/kbfiltr | 94 | 4228 | × | No | × | No | × | Yes | 2m 53s | 5 |
| src/general/toaster/toastmon | 123 | 6261 | FP | - | FP | - | √ | - | 4m 1s | 3 |
| src/storage/filters/diskperf | 121 | 6584 | FP | - | FP | - | √ | - | 3m 17s | 3 |
| src/network/modem/fakemodem | 142 | 8747 | FP | - | FP | - | √ | - | 11m 6s | 3 |
| src/storage/fdc/flpydisk | 171 | 12752 | FP | - | FP | - | FP | - | 1h 6m | 5 |
| src/input/mouclass | 192 | 13380 | FP | - | FP | - | FP | - | 40m 26s | 5 |
| src/input/mouser | 188 | 13989 | FP | - | FP | - | FP | - | 1h 4m | 5 |
| src/kernel/serenum | 184 | 14123 | FP | - | FP | - | √ | - | 19m 41s | 2 |
| src/wdm/1394/driver/1394diag | 171 | 23430 | FP | - | FP | - | FP | - | 1h33m | 5 |
| src/wdm/1394/driver/1394vdev | 173 | 23456 | FP | - | FP | - | FP | - | 1h38m | 5 |

**Table 2.** Results of checking the *PendedCompletedRequest* rule in Windows device drivers. (√: passes rule; ×: a real bug found; FP: false positive.) See Tab. 1 for an explanation of $\ominus$, $\circledcirc$, and $\bigstar$. (For the examples that pass the rule, "Rounds" represents the number of VSA-ASI rounds required to prove the absence of the bug; for the other examples, the maximum number of rounds was set to 5.)

```
int dispatch_routine(...) {
  int status, c = 0;
  .
  .
  status = STATUS_PENDING;
  P1:if(...) {
    status = STATUS_SUCCESS;
    c = 1;
  }
  P2:
  .
  .
  if(c == 1) {
    IoCompleteRequest(...)
  }
  P3: return status;
}
```

```
Information at P3 with the FSM shown in Fig. 4
START:
    c ↦ {0, 1}
    status ↦ {STATUS_SUCCESS, STATUS_PENDING}
COMPLETED:
    c ↦ {0, 1}
    status ↦ {STATUS_SUCCESS, STATUS_PENDING}
```

```
Information at P3 with the FSM shown in Fig. 6
ST_PENDING:
    c ↦ {0}
    status ↦ {STATUS_PENDING}
ST_NOT_PENDING:
    c ↦ {1}
    status ↦ {STATUS_SUCCESS}
```

**Fig. 5.** An example illustrating false positives in device-driver analysis.

necessary to determine the status-variable for each procedure. However, because debugging information is usually not available, we use the following heuristic to identify the status-variable for each procedure in the executable:

> *By convention,* `eax` *holds the return value in the x86 architecture. Therefore, the local variable (if any) that is used to initialize the value of* `eax` *just before returning from the dispatch routine is considered to be the status-variable.*

Configuration '$\bigstar$' uses the automaton obtained by combining the *PendedCompletedRequest* FSM and the FSM shown in Fig. 6 (instantiated using the above heuristic) using a cross-product construction. As shown in Tab. 2, for configuration '$\bigstar$', the number of false positives is substantially reduced.

It required substantial manual effort to find an abstraction that had sufficient fidelity to reduce the number of false positives reported by DDA/x86. To create a practical tool, it would be important to automate the process of refining the abstraction based on the property be checked. The model-checking community has developed many techniques that could be applicable, although the discussion above shows that the definition of a suitable refinement can be quite subtle.

As a point of comparison, SDV also found the bugs in both "moufiltr" and "kbfiltr", but had no false positives in any of the examples. However, one should not leap to the conclusion that machine-code-analysis tools are necessarily inferior to source-code-analysis tools.

- The basic capabilities are different: DDA/x86 can analyze stripped device-driver executables, which goes beyond the capabilities of SDV.
- The analysis techniques used in SDV and in DDA/x86 are incomparable: SDV uses predicate-abstraction-based abstractions [21], plus abstraction refinement; DDA/x86 uses a combined numeric-plus-pointer analysis [5], together with a different kind of abstraction refinement [6]. Thus, there may be examples for which DDA/x86 outperforms SDV.

Moreover, SDV is a multiple man-year effort, with a professional team at Microsoft devoted to its development. In contrast, the prototype DDA/x86 was created in only a few man-months (although multiple man-years went into building the underlying CodeSurfer/x86 infrastructure).

**Property Automata for the Analysis of Machine Code.** Property automata for the analysis of machine code differ from the automata used for source-level analysis. In particular, the domain of discourse—the alphabet of actions to which an automaton responds—is different when property checking is performed at the machine-code level, rather than on an IR created from source code.

In some cases, it is possible to recognize a source-level action based on information available in the recovered IR. For instance, a source-code procedure call with actual parameters is usually implemented as a sequence of instructions that evaluate the actuals, followed by a `call` instruction to transfer control to the starting address of the procedure. The IR-recovery algorithms used in CodeSurfer/x86 will identify the call along with its arguments.

In other cases, a source-level action is not identifiable. One contributing factor is that a source-level action can correspond to a sequence of instructions. Moreover, the instruction sequences for two source-level actions could be interleaved. We did not have a systematic way to cope with such problems except to rewrite the automaton of interest based on instruction-level actions.

Fortunately, most of the instruction-level actions that need to be tracked boil down to memory accesses/updates. Because VSA is precise enough to interpret many memory accesses [4, §7.5], it is possible for DDA/x86 to perform property checking using the extended version of VSA described in §3. In our somewhat limited experience, we found that for many property automata it is possible to rewrite them based on memory accesses/updates so that they can be used for the analysis of executables.

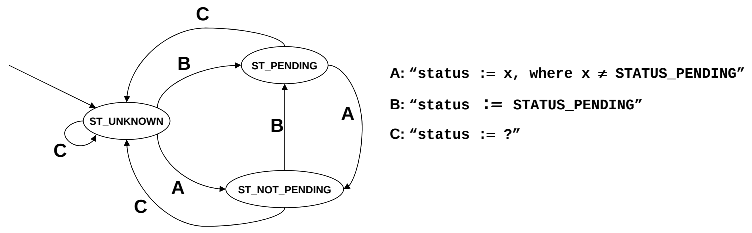
C
B
ST_PENDING
ST_UNKNOWN
B
A
C
A
ST_NOT_PENDING
C
A: "status := x, where x ≠ STATUS_PENDING"
B: "status := STATUS_PENDING"
C: "status := ?"

**Fig. 6.** Finite-state machine that tracks the contents of the variable `status`.

**Finding a Witness Trace.** If the VSA algorithm reports that the `ERROR` state in the property automaton is reachable, it is useful to find a sequence of instructions that shows how the property automaton can be driven to `ERROR`. Rather than extending the VSA implementation to generate and manage explicitly the information required for reporting witness traces, we exploited the fact that the standard algorithms for solving reachability problems in pushdown systems (PDSs) [14, 19] provide a witness-trace capability to show how a given (reachable) configuration is reachable.

The algorithm described in §3 was augmented to emit the rules of a PDS on-the-fly. The PDS constructed is equivalent to a PDS that would be obtained by a cross-product of the property automaton and a PDS that models the interprocedural control-flow graph, except that, by emitting the PDS on-the-fly as VSA variant '★' is run, the cross-product PDS is pruned according to what the VSA algorithm and the property automaton both agree on as being reachable. The PDS is constructed as follows:

| PDS rules | Control flow modeled |
|---|---|
| $\langle q, [n_0, s]\rangle \hookrightarrow \langle q, [n_1, s']\rangle$ | Intraprocedural CFG edge from node $n_0$ in state $s$ to node $n_1$ in state $s'$ |
| $\langle q, [c, s]\rangle \hookrightarrow \langle q, [enter_{\mathtt{P}}, s][r, s']\rangle$ <br> $\langle q_{[x_{\mathtt{P}}, s']}, [r, s']\rangle \hookrightarrow \langle q, [r, s']\rangle$ | Call to procedure `P` from $c$ in state $s$ that returns to $r$ in state $s'$. |
| $\langle q, [x_{\mathtt{P}}, s']\rangle \hookrightarrow \langle q_{[x_{\mathtt{P}}, s']}, \epsilon\rangle$ | Return from `P` at exit node $x_{\mathtt{P}}$ in state $s'$ |

In our case, to obtain a witness trace, we merely use the witness trace returned by the PDS reachability algorithm to determine if a PDS configuration $\langle q, [n, \mathtt{ERROR}]\rangle$—where $n$ is a node in the interprocedural CFG—is reachable from the configuration $\langle q, \mathtt{enter}_{\mathtt{main}}\rangle$.

Because the PDS used for reachability queries is based on the results of VSA, which computes an *over-approximation* of the set of reachable concrete memory states, the witness traces provided by the reachability algorithm may be infeasible. In our experiments, only for configuration '★' were the witness traces for `kbfiltr` and `moufiltr` feasible. (Feasibility was checked by hand.)

This approach is not specific to VSA; it can be applied to essentially any worklist-based dataflow-analysis algorithm when it is extended with a property automaton, and provides a conceptually low-cost mechanism for augmenting such algorithms to provide witness traces.

## 5 Related Work

DDA/x86 is the first known application of program analysis/verification techniques to stripped industrial executables. Among other techniques, it combines the IR-recovery algorithms from CodeSurfer/x86 [4–6] with the path-sensitive method of interpreting property automata from ESP [17].

A number of algorithms have been proposed in the past for verifying properties of programs when source code is available [10, 12, 13, 17, 20, 22]. Among these techniques, SDV [10, 12] and ESP [17] are closely related to DDA/x86. SDV builds a Boolean representation of the program using predicate abstraction; it reports a possible property violation if an error state is reachable in the Boolean model. In contrast, DDA/x86 uses value-set analysis [5, 4] (along with property simulation) to over-approximate the set of reachable states; it reports a possible property violation if the error state is reachable at any instruction in the executable. To eliminate spurious error traces, SDV uses counter-example-guided abstraction refinement, whereas DDA/x86 leverages path sensitivity obtained by combining property simulation and abstract interpretation. In this respect, DDA/x86 is more closely related to ESP—in fact, the algorithm in §3 was inspired by ESP. However, unlike ESP, DDA/x86 provides a witness trace for a possible bug, as described in §4. Moreover, DDA/x86 uses a different kind of abstraction refinement [6].

Although combining the propagation of property-automaton states and abstract interpretation provides a degree of path sensitivity, it was not always sufficient to eliminate all of the false positives for the examples in our test suite. Therefore, we also distinguished paths based on the abstract state (using the automaton shown in Fig. 6) in addition to distinguishing paths based on property-automaton states. While the results of our experiments are encouraging, it required a lot of manual effort to reduce the number of false positives: spurious error traces were examined by hand, and the automaton in Fig. 6 was introduced to refine the abstraction in use. For DDA/x86 to be usable on a day-to-day basis, it would be important to automate the process of reducing the number of false positives. Several techniques have been proposed to reduce the number of false positives in abstract interpretation, including trace partitioning [26], qualified dataflow analysis [23], and the refinement techniques of Fisher et al. [20] and Dhurjati et al. [18]. All of these techniques are potentially applicable in DDA/x86.

## References

1. http://www.microsoft.com/whdc/devtools/ddk/default.mspx.
2. Defrauding the WHQL driver certification process, March 2004. http://blogs.msdn.com/oldnewthing/archive/2004/03/05/84469.aspx.
3. C++ for kernel mode drivers: Pros and cons, February 2007. WHDC web site, http://www.microsoft.com/whdc/driver/kernel/KMcode.mspx.
4. G. Balakrishnan. *WYSINWYX: What You See Is Not What You eXecute*. PhD thesis, C.S. Dept., Univ. of Wisconsin, Madison, WI, August 2007. TR-1603.

5. G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *CC*, 2004.
6. G. Balakrishnan and T. Reps. DIVINE: DIscovering Variables IN Executables. In *VMCAI*, 2007.
7. G. Balakrishnan, T. Reps, N. Kidd, A. Lal, J. Lim, D. Melski, R. Gruian, S. Yong, C.-H. Chen, and T. Teitelbaum. Model checking x86 executables with CodeSurfer/x86 and WPDS++. In *CAV*, 2005.
8. G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. WYSINWYX: What You See Is Not What You eXecute. In *IFIP Working Conf. on VSTTE*, 2005.
9. T. Ball. Personal communication, February 2006.
10. 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 *EuroSys*, 2006.
11. T. Ball and S.K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *Spin Workshop*, 2000.
12. T. Ball and S.K. Rajamani. The SLAM toolkit. In *CAV*, 2001.
13. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, 2003.
14. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *CONCUR*, 1997.
15. A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *SOSP*, 2001.
16. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, 1979.
17. M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI*, 2002.
18. D. Dhurjati, M. Das, and Y. Yang. Path-sensitive dataflow analysis with iterative refinement. In *SAS*, pages 425–442, 2006.
19. A. Finkel, B.Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *Elec. Notes in Theor. Comp. Sci.*, 9, 1997.
20. J. Fischer, R. Jhala, and R. Majumdar. Joining dataflow with predicates. In *FSE*, 2005.
21. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, 1997.
22. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.
23. L.H. Holley and B.K. Rosen. Qualified data flow problems. *TSE*, 7(1):60–78, 1981.
24. IDAPro disassembler, http://www.datarescue.com/idabase/.
25. J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *CC*, 1992.
26. L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *ESOP*, 2005.
27. W. Oney. *Programming the Microsoft Windows Driver Model*. Microsoft, 2003.
28. G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *POPL*, 1999.
29. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.
30. M.M. Swift, M. Annamalai, B.N. Bershad, and H.M. Levy. Recovering device drivers. In *OSDI*, 2004.
31. M.M. Swift, B.N. Bershad, and H.M. Levy. Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst.*, 23(1), 2005.