Formal Methods in Computer-Aided Design 2020

Angelic Checking within Static Driver Verifier: Towards high-precision defects without (modeling) cost

Shuvendu K. Lahiri*, Akash Lal*, Sridhar Gopinath§, Alexander Nutz‡, Vladimir Levin†, Rahul Kumar*, Nate Deisinger†, Jakob Lichtenberg†, and Chetan Bansal*
*Microsoft Research
†Microsoft
‡University of Freiburg
§University of Texas at Austin

***Abstract*—Microsoft's Static Driver Verifier (SDV) pioneered the use of software model checking for ensuring that device drivers correctly use operating system (OS) APIs. However, the verification methodology has been difficult to extend in order to support either (a) new classes of drivers for which SDV does not already have a harness and stubs, or (b) memory-corruption properties. Any attempt to apply SDV out-of-the-box results in either *false alarms* due to the lack of environment modeling, or *scalability issues* when finding deeply nested bugs in the presence of a very large number of memory accesses.**

In this paper, we describe our experience designing and shipping a new class of checks known as *angelic checks* through SDV with the aid of *angelic verification* (AV) [1] technology, over a period of 4 years. AV pairs a precise inter-procedural assertion checker with automatic inference of *likely specifications* for the environment. AV helps compensate for the lack of environment modeling and regains scalability by making it possible to find deeply nested bugs, even for complex memory-corruption properties. These new rules have together found over a hundred confirmed defects during internal deployment at Microsoft, including several previously unknown high-impact potential security vulnerabilities. AV considerably increases the reach of SDV, both in terms of drivers as well as rules that it can support effectively.

I. Introduction

Microsoft's Static Driver Verifier (SDV) [2], [3] is a formal software verification tool that checks Windows device drivers against a set of rules on the correct usage of operating system (OS) APIs. These Windows OS APIs, which are published on MSDN, and exported for writing Windows drivers are commonly referred to as the Driver Development Interface (DDI) [4]. SDV is shipped to driver developers in the Windows ecosystem through the Windows Driver Development Kit (WDK). Running SDV is a mandated check for a driver to obtain certification for Windows Server OS [5].

Examples of SDV rules range from checking that a driver calls a DDI function at a particular Interrupt Request Level (IRQL), to ensuring that locks are acquired and released in correct sequence. Given: (a) a rule, written in SDV's specification language SLIC [6], (b) a *harness* for the driver class (e.g. storage or networking) that determines how the driver can be invoked by the OS, and (c) *stubs* for OS DDI functions that can be invoked by the driver, SDV uses a software verification tool to look for driver executions that violate the rule. SDV also has a detailed defect viewer to aid in debugging. The viewer allows stepping through (interprocedural) counterexample traces reported by the verifier. The trace contains not just control-flow information but also values of various variables along the trace. The verification "engine" powering the analysis has transformed from SLAM [7] to SLAM-2 [8] to YOGI [9] and then finally to Corral [10], [11], each time improving performance, accuracy and scalability. SDV establishes a high bar for precision of each of its supported rules; typically false-positives rate is below 5% [8].

However, despite a decade of investment in the technology and advances in the underlying verification engines [10], [12], it has been difficult to adapt the tool to new verification challenges, even within the world of device drivers. Specifically,

- **Memory safety:** Checking for memory corruption bugs within the driver code.
- **Unsupported drivers:** Performing verification of DDI compliance rules for unsupported driver classes whose frameworks are not modeled accurately by harnesses and stubs.

Memory safety violation is broadly interpreted as an unchecked invalid access to a piece of memory during a program's execution. These issues are prevalent in low-level languages such as C and C++ that trade off performance for automatic memory management overheads. Memory safety violations can be either (i) *temporal*, which pertains to a type-state on a memory address such as being `allocated`, `freed`, or `null`, or (ii) *spatial*, which pertains to checking bounds of an allocated buffer. Such violations can have serious implications on both the reliability and, more importantly, on the security of the entire system. Recent studies attribute as much as 70% of all security bugs in Microsoft products can be attributed to memory safety issues [13]. Static analysis tools such as SDV are particularly attractive for the domain of kernel-mode drivers due to the poor coverage of dynamic tools in this space (because of System issues in setting up dynamic tools for kernel-mode components, as well as the large input space).

On the other hand, SDV currently supports environments for certain general purpose drivers (WDM, WDF) as well as two driver classes (storage, networking). However, there are several important driver classes that SDV doesn't support, including file system filters, audio drivers, kernel streaming (in particular, camera) drivers. For such drivers, SDV environment is insufficient, which results in partial coverage at best (lack of a harness that calls the entry points) and false alarms (as DDIs implemented by a driver class library are not modeled by stubs).

In other words, SDV enjoys a very high *precision* on the rules and driver classes it supports, but has a relatively poor *recall* (or *coverage*) on the set of all possible bugs discovered in Windows drivers, particularly those that affect memory safety. In this paper, we therefore focus on the following problem in the context of SDV:

> *Can we improve the coverage (or recall) of reliability and security bugs in Windows drivers using SDV without sacrificing the high precision bar?*

Note that retaining the high precision bar is necessary for SDV; customers will push back if SDV slowed down the development process due to the need to deal with spurious alarms. We are careful to pose the problem as that of "improving the coverage" rather than "full coverage" (or ensuring the absence of) of newer class of defects not detected by SDV. At the same time, it is desirable that the approach is able to leverage additional modeling (if present) to achieve higher coverage[1].

We studied the main technical difficulties to adapt SDV to new verification challenges, and narrowed it down to a combination of two reasons:

1) **Precision due to un(der)-constrained environment models:** Creating models for a new class of drivers requires upfront investment that ranges from several weeks to months of effort and close interaction with domain experts. Furthermore, even the existing models for DDIs may leave most behaviors unspecified, focussing on just the ones that matter for the current set of rules. This is especially troublesome for memory-safety rules because existing models leave out pointer-related behaviors. For example, it was very common for existing stubs to non-deterministically return a `null` pointer as output; this was fine for existing SDV rules, until we started checking for `null`-safety and got numerous false alarms.
2) **Scalability:** Software verifiers face a path-explosion problem when large parts of a program cannot be abstracted with *summaries*. Although there has been progress in performing modular software verification for simple properties, performing summarization for memory safety with SMT solvers is still an open research problem. This is primarily due to the need to summarize the state of the heap including state in linked data structures. Secondly, the path explosion problem worsens with the depth of nesting of procedure calls. Even when a bug is *localized* to a procedure, e.g., the procedure sets a pointer to `null` and then dereferences it, the verifier could still fail (timeout) in trying to enumerate feasible paths from a driver entrypoint to the procedure. This search is unnecessary because a user can immediately identify the bug without looking at the rest of the code. This indicates a shortcoming of the SDV approach.

[1]One can view this as the principle of *pay-as-you-go verification*.

Rule	Bugs
NULLCHECK	68
USEAFTERFREE	7
DOUBLEFETCH	11
IRQLCHECK	26
Total	112

Fig. 1. A count of true bugs found by SDV using angelic checks.

This paper describes how we significantly extended the reach of SDV through a set of *angelic checks*. We distinguish these checks from the currently supported (*demonic*) checks, where the expectation was that SDV does due diligence to provide accurate harnesses and stubs with full path coverage for loop-free programs. We observed that we can address both the issues of (a) spurious alarms from under-constrained environment, and (b) scalability to find defects in deeply-nested methods, by using angelic verification technology [14], [1].

Angelic verification equips a precise *interprocedural* verifier (such as Corral) with automatic inference of *likely specifications* for the unknowns that correspond to values controlled by the environment. AV suppresses alarms from the verifier if it can infer a *reasonable* environment specification to rule out the alarm. Furthermore, because AV can also tolerate an unconstrained initial state, AV can start exploration from any driver procedure, not just the harness. This is beneficial for catching deeply-nested bugs.

Since the AV technology works on programs written in Boogie [15], we also developed an instrumentation framework called AVP for Boogie programs that we used to instrument the new class of memory safety properties. We have used the new framework to successfully add several angelic checks to SDV with the goal of realizing the above vision. These include the following:

1) NULLCHECK: checks that a `null`-valued pointer is not dereferenced,
2) USEAFTERFREE: checks that a freed pointer is not used (or freed again),
3) DOUBLEFETCH: checks that a *userland* pointer is not dereferenced twice in any execution within the kernel [16],
4) IRQLCHECK: checks that DDI calls are made only at an acceptable IRQL state.

The first three rules above pertain to memory safety (or memory corruption) rules. Moreover, a violation of USEAFTERFREE and DOUBLEFETCH rules can expose a serious security vulnerability.

We report our experience developing these checks and specifically comment on the trade-offs between increasing recall at the cost of sacrificing precision. These rules have been evaluated on close to a thousand drivers within Microsoft, including drivers outside SDV's supported list. These new rules have together found over a hundred confirmed defects during an internal deployment at Microsoft, including several

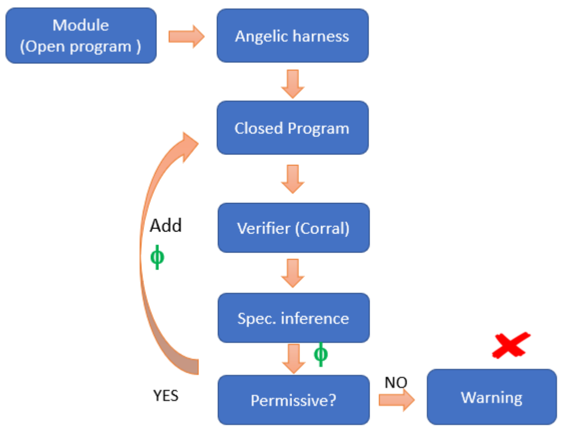

Fig. 2. AV tool flow

```
int Foo(int *x, int *y,
        bool f)
{
   *x = 1;
   Bar(x, y);
   if (f) { free(x); }
   ...
}
void Bar(int *x, int *y)
{
    free(x);
    *y = 2;
}
void free (int *x);
```

```
procedure Foo(x: int, y: int,
              f: bool)
{
   assert (!Freed[x]);
   Mem[x] := 1;
   call Bar(x, y);
   if (f) { call free(x); }
   ...
}
procedure Bar(x: int, y: int)
{
   call free(x);
   assert (!Freed[y]);
   Mem[y] := 2;
}
procedure free(x: int)
{
    assert (!Freed[x]);
    Freed[x] := true;
}
```

Fig. 3. A program in C and its encoding in Boogie.

previously unknown high-impact potential security vulnerabilities. The exact counts are summarized in Figure 1. All bugs, except one for DOUBLEFETCH, were previously unknown. AV was able to suppress tens of thousands of spurious traces in total over all these examples, indicating that these bugs could not have been discovered without the angelic checks.

At the time of writing this paper, all the rules except DOUBLEFETCH are available with Windows 10 WDK. The DOUBLEFETCH rule is currently a part of *preview* versions of the WDK for co-engineering partners. Furthermore, we note that the AV tool is available open-source[2]. We hope that the experience captured in this paper, in conjunction with open-source AV, allows the development of similar tools in domains other than device drivers.

The rest of the paper is organized as follows. Section II presents background on the AV technology. Section III describes the angelic checks for memory safety, which incudes NULLCHECK, USEAFTERFREE as well as DOUBLEFETCH. Section IV covers the role of IRQL in device drivers and the corresponding angelic check for it (IRQLCHECK). Section V describes related work and Section VI concludes.

II. ANGELIC CHECKS

In this section, we describe background on angelic verification (Section II-A) and some details of the AVP property instrumentation language (Section II-B).

A. Angelic Verification

Angelic verification [1] (AV) is a technique for leveraging automatic static assertion checkers for finding high-confidence defects in open programs. The technique pairs a precise assertion checker (AC) (that can find interprocedural traces for assertion violations in closed programs) with the inference of *angelic specifications* on the environment. The latter is used to push back on the AC verifier from reporting “dumb” false alarms in open programs.

[2]https://github.com/boogie-org/corral/tree/master/AddOns/AngelicVerifierNull

Figure 2 describes the high-level flow of the algorithm. Given an open program with a set of assertions in Boogie [15], we first close the program with an *angelic harness*. The angelic harness helps to create a unified representation of unknown values resulting from both the input state (value of parameters and the heap state when program execution begins) as well as the output state of an external call (return value as well as side-effects). We refer the readers to earlier work for further details [1]. The harness non-deterministically calls into all procedures of the input program. (For the purpose of this section, we make the simplifying assumption that the program contains no external methods.) AV invokes a whole-program verifier (CORRAL) in a loop to enumerate traces that violate an assertion in the input program. For each such failure trace τ starting at a procedure p with unconstrained inputs over X that violates an assertion, AV infers a precondition ϕ over X that ensures $\phi \Rightarrow wp(true, \tau)$, where *wp* stands for the *weakest liberal precondition* [17]. AV then checks if the precondition is consistent with the previously inferred specifications (starting with *true*). If so, it suppresses the trace τ, else it marks τ as an *angelic trace* to be displayed to the user.

AV provides several *knobs* to the rule developer in order to control the expressiveness of the inferred specifications ϕ, which in turn helps determine the scalability and the precision of AV on that rule. Among other things, an angelic check is parameterized by a *vocabulary* V of predicates that constitutes the atoms in the preconditions (the pool of candidate predicates are automatically mined from the trace). For example, we can require the vocabulary to only consist of non-aliasing predicates, connected with arbitrary Boolean connectives. Further, AV allows the analysis to only suppress the *data flow* (i.e. consider *wp* while treating all conditionals in a path as non-deterministic) or consider the *control flow* of the defect as well [14], [1]. For the checks presented in this paper, AV only considered blocking the data flow.

Figure 3 shows AV’s working on a simplified version of the USEAFTERFREE rule. The figure shows the program in C, as well as its encoding in Boogie, where the heap is modeled using an array `Mem` that maps a pointer to its contents. (A more detailed explanation on the encoding of C semantics in Boogie can be found in previous work [18], [11].) The underlined statements in the Boogie encoding denote the instrumentation performed for checking USEAFTERFREE on the code. This instrumentation happens via the AVP tool described in Section II-B. We introduce a map `Freed` to track the allocated-ness of a pointer, and add assertions before the use of a pointer (either a dereference or a free of the pointer).

AV requires two verification queries for this program, one that starts program execution at `Foo` and other that starts at `Bar`. The analysis of `Bar` produces two error traces due to unconstrained inputs, which can be blocked using preconditions `!Freed[x]` and `!Freed[y] && x != y` respectively. Note the role of the vocabulary V here; if we disallow equality predicates in our vocabulary, then there are no permissive specifications to block the second trace. However, the only specification for `Foo` to block the trace that frees `x` twice by descending into `Bar` is `!f`, which creates dead code and is not permissive. For the angelic checks in SDV, we decided to not block the trace based on control flow as described earlier. Thus, in this case, AV will report one angelic trace for `Foo` that will be displayed to the user. Next we briefly describe the property instrumentation tool for creating the input to AV.

B. AVP Instrumentation Language

Since AV operates on Boogie programs, we designed a custom domain-specific language (DSL) called AVP[3] that describes a source-to-source instrumentation of Boogie programs. The language is a collection of *LHS-to-RHS* rules. A rule can pattern-match on Boogie AST nodes like expressions, statements or procedures and present a rewriting of the match.

Each angelic check is described as an AVP file whose purpose is to add ghost state (such as `Freed` in Figure 3) and instrument the necessary assertions and updates to ghost variables into the program. The NULLCHECK rule, for instance, matches on the base pointer p of a dereference (such as `*(p+4))` and adds the following assertion right before the dereference:

```
assert (!aliases(p, NULL) || p != NULL)
```

The `aliases` function triggers AV's alias analysis as a pre-pass. If the analysis finds that an expression `e1` cannot alias `e2`, then it replaces the occurrence `aliases(e1, e2)` syntactically with `false`; otherwise, it is replaced with `true`. The ability to refer to alias analysis allows us to express not just syntactic, but a more semantic program instrumentation to add a property. We leverage this feature for all the memory safety properties. But it is important to note that we use alias analysis only as an optimization to prune the space of assertions. It does not affect precision given an interprocedural checker that reasons precisely about aliasing within the module and the specification inference takes care of possible spurious aliasing due to the environment.

[3]https://github.com/boogie-org/corral/wiki/AV-Property-(AVP)-Language

III. ANGELIC MEMORY SAFETY CHECKS

In this section, we describe the different angelic checks related to memory safety. Recall that such issues arise primarily in low-level languages such as C and C++ that rely on the programmer to ensure that a program does not access invalid memory. Examples of invalid accesses include: accessing a `null` pointer, accessing a pointer after it has been freed, or accessing a pointer outside the bounds of an allocated object. In recent years, many of these invalid memory accesses have lead to security exploits, where an attacker can trick a system to perform information leak or remote code execution [19], [13]. Some of the classical memory safety issues can be mitigated by programming in *managed* languages such as Rust, where accesses to invalid addresses lead to runtime exceptions with clear semantics (unlike the undefined behaviors for programs written in *unmanaged* languages). However, other security relevant memory safety issues (e.g. DOUBLEFETCH) that result from the kernel-user boundary [16], [20], [21], [22] may be applicable even when the drivers are authored in memory safe languages such as Rust.

In this section, we describe the different rules and our experience with deploying them. These properties were developed and tested over various points over the course of four years, so we report our evaluation of the rules as they were developed and tested before being rolled out to customers. Note that we did not change the internals of AV tool for supporting these multiple properties. Each property is completely contained in its own AVP file.

A. Nullcheck

In our earlier work [1], we presented an evaluation of the angelic NULLCHECK rule on 10 modules in Windows, totaling over 300K lines of code, and compared it with a mature tool PREfix. We briefly summarize our findings. PREfix is an industrial strength tool being used at Microsoft for over a decade, and has custom algorithms for null-checking as well as accurate models for many OS components. Over a set of 68 defects that PREfix reported in these 10 modules, we managed to confirm over 80% of the defects, found several false alarms in the PREfix defects due to imprecise modeling of C semantics, and also discovered new true alarms not reported by PREfix. AV also reported less than 10% of false positives on this set (leaving aside frontend translation issues that have subsequently been addressed).

These results were seen as encouraging by the SDV product team and lead to the integration of NULLCHECK as the first angelic check in SDV. The rule now appears documented on MSDN[4] since its release in 2018. This section outlines the further insights that were needed to take NULLCHECK from a research prototype to a push-button check available to the entire Windows driver ecosystem.

First, we performed several improvements on the precision and usability of the check when evaluating on Windows drivers. The chief among them are the following.

[4]https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/nullcheckw?redirectedfrom=MSDN

1) **Improved alias analysis:** We designed an alias analysis that works on a Boogie program and use it to prune away `null` checks on pointers that cannot alias `null`. The analysis implements the usual Andersen's may-alias analysis [23], however it is optimized to track the flow of `null`-value very precisely [24]. The analysis is able to prune away almost 98% of all `null`-checks.
2) **Changes to existing SDV models:** We needed to modify SDV's harness and stubs even for supported driver frameworks. These modifications consisted of the following major changes. (a) First, we found that several existing OS models mistakenly left an output pointer unallocated even when the return `NT_STATUS` error code denoted successful allocation. This leads to a false alarm because AV will discover a `null`-dereference in the caller; the alarm cannot be suppressed without creating dead code. (b) Second, we did not use SDV's harness even when present. Recall that a harness calls the driver APIs in a sequence in order to mimic how an actual OS would invoke the driver. Using the harness would cause the analysis to time out even when driver had simple bugs but were embedded deep inside the driver. Instead of using the harness, we run AV on all methods of the driver in parallel. This allows SDV to get much better coverage on the driver. However, dropping the harness does cause us to miss bugs that can only be triggered by calling several driver APIs in sequence.
3) **Side-effects of external methods:** Finally, we realized that AV suppresses spurious alarms arising from external methods only when it can estimate the side-effects of the external method. We currently use the signature of an external method to automatically determine the side effects in addition to the return value. For example, if an external method takes an argument e of static type `int **`, then we assume that the values of $*e$ and $**e$ may be modified. This in turn allows AV to infer angelic specifications on these unknown modified values. However, this heuristic does not work when the external method (a) modifies some global variable, or (b) exits under some condition. Of these, the former happens frequently when the external method sets up a global function pointer that is invoked later in the driver. We, therefore, manually added models for such DDI functions driven by the false alarms we saw during our evaluation.

We evaluated the NULLCHECK rule on a set of 192 real-world drivers that constitute the *Integration Test Pass* (ITP), the regression test suite for the SDV product. These constitute drivers from *Storport*, *KMDF*, *WDM* and *NDIS* classes of drivers. Of these, we found 61 defects over 27 drivers and 3 drivers timed out after 3000 seconds. Several of the authors spent a few months (of 2016) to inspect these traces, and consulted with driver experts to determine the ground truth on their validity. We finally determined that 58 out of the 61 defects were *true defects* (95% precision). Several of these bugs (after removing duplicates that require the same underlying fix) were filed and fixed. The bugs that were confirmed but not fixed mainly came from two categories: (a) the driver was no longer being maintained or shipped externally, or (b) there was a runtime assertion `NT_ASSERT(FALSE)` in debug mode that would cause the driver to crash if the pointer was `null`, eliminating any security implications (12 such cases). These runtime assertions indicate that the driver developers suspected these pointers could be `null` at runtime although there is no proof of their absence. The 3 false alarms came from the absence of two models of external functions and 1 modeling issue for C arrays in the front end; the latter has since been fixed. The two OS models required modeling of linked lists `ExInterlockedRemoveHeadList` and additional ghost state in `IoAttachDeviceToDeviceStack`, neither of which were deemed cost effective to add.

In addition to the 58 bugs in ITP, at least 10 more true bugs have been found and confirmed by SDV team during internal deployment. The rule has found a couple of potential security bugs in Windows drivers, of which at least one was classified by a security review as critical, (hence) immediately fixed and the fix was taken to an OS security update (in 2016).

B. Use After Free

Figure 3 showed an example of the USEAFTERFREE angelic check. The ability to use a pointer after it has been freed has serious implications ranging from corruption of valid data to remote code execution vulnerabilities [25]. In this section, we describe the rule in more detail and present our experience with deploying it internally in Microsoft.

The rule ensures that a non-null pointer that has been *freed* by calling a DDI function (either `free` or variants such as `IoFreeMdl`, etc.), is not *used*[5]. A pointer is used if it is an argument to a routine that frees the pointer (special case signifying *double-free*), or is dereferenced. To specify the rule, we leverage our alias analysis to guard the check to only those pointers that can potentially be aliased with a freed-pointer within the module. This is achieved by tracking a global variable `freedp` that can be non-deterministically assigned one of the pointers that is freed, and weakening the assertion to only consider pointers that could be aliased with `freedp`. We refined the rule to allow freeing of `null`, which is a valid behavior. In fact, it is a common practice to set a freed pointer to `null`, and not check for a pointer to be non-null before freeing it.

We performed an extensive evaluation on 65 drivers that contain at least one call to a method named "free". Our initial rule was more aggressive than the final rule described above in two aspects to not miss defects during evaluation:

1) We considered any external method with a substring *free* as a method that could potentially free a pointer, and
2) We considered a pointer argument to any external procedure as a use; our intuition was that it is a bad practice to pass a freed pointer externally.

[5]The property file is located here: https://github.com/boogie-org/corral/blob/master/AddOns/PropInst/PropInst/ExampleProperties/useafterfree-razzle.avp

```
void Foo(x) {
   Increment(&x->RefCnt);
   Bar(x);    // may free x
   x->f = 1; // use
}
```

```
void Bar() {
   Decrement(x->RefCnt);
   if(x->RefCnt == 0) {
       free(x);
   }
}
```

Fig. 4. False alarm for UseAfterFree.

We obtained a total of 69 traces in 22 drivers that we carefully inspected after removing duplicates. We filed 7 new bugs with developers and most of them were either fixed or confirmed but not fixed due to the lack of support for the driver. Many of these bugs were due to cleanups along exception paths, making them difficult to reach during regular testing.

A majority of the spurious alarms resulted from the two decisions above. For example, routines such as `RtlFreeUnicodeString` take a pointer to a structure as an argument, but only free the `Buffer` field of that structure. Similarly, we found several low-level print functions such as `TracePrint` that never dereference the pointer and therefore safe. In addition to these, there was one class of false alarms in 3 traces that demonstrates a fundamental limitation of the angelic choice that we adopted. We discuss this in more detail.

Consider a simplified program in Figure 4. AV performs two checks, starting at each of the two procedures. When analyzing `Foo`, there is a path where `x` is freed and later dereferenced. This is a false alarm as callers of `Foo` ensure that `x->RefCount` is always greater than 1 on entry. However, the default configuration of AV does not infer a specification on `RefCount` values because we treat path conditions as non-deterministic. In other words, we only treat the data-flow in an angelic manner, not the control flow. We are working on improving AV to push the traces all the way to module entry points in such cases to remove this class of false alarms.

C. Double Fetch

Consider the method `Foo` below that marks an entry point into the kernel and consider a pointer parameter `x` that can be controlled by the user from a user-mode application or driver (referred to as *userland* pointers). Consider an execution when the pointer is "fetched" twice, in lines 2 and 5.

```
1   NT_STATUS Foo(A *x, ..) {
2       int len = x->Length;
3       if (len > 0) {
4          char *y = malloc(len);
5          RtlCopyMemory(y, x->Length, x->Buffer);
6       }
7   }
```

A malicious user may alter the value of `x->Length` between the two lines, resulting in a buffer overflow of the kernel memory, which can be exploited for information disclosure or remote code execution. These bugs have lead to several security vulnerabilities in both Linux and Windows kernels and therefore of great concern to kernel-mode driver developers.

The double-fetch property is encoded as an angelic check[6]. Similar to UseAfterFree, we maintain a map `hasBeenRead` that maps each address to the number of times it has been read, and ensure that a userland pointer is never read twice. However, *userland* pointers cannot be distinguished from kernel-allocated pointers without precisely marking the kernel entry points. Instead, we approximate it by assuming that that driver developer at least *probes* userland pointers before accessing them in the kernel. We use a similar trick as UseAfterFree where we consider a pointer as userland if it aliases with the argument of either `ProbeForRead` or `ProbeForWrite`.

We have currently evaluated this rule on one driver where a violation was detected by security pen-testers manually over a year ago. The bug was present in a C++ module, and spanned several method calls between the site of probe and the two fetches. Further, one of the fetches was directly inside a condition statement. To recover the bug, we had to fix some issues in the SDV front end for C++ as well as set a high timeout of 9000 seconds for the angelic check. Not only did we recover the *precise* bug, but we also discovered 10 more variants of the bug (all confirmed by the pen-testers) on other pointers and procedures on the same version of the driver (the driver was already rewritten substantially after previous bugs). There has not been any false alarms from this rule to date on either this or other drivers that we have tested so far, although further evaluation starting with the ITP suite is still pending. The DoubleFetch rule is currently a part of preview versions of the WDK for co-engineering partners, and will be made available to broader ecosystem in the near future.

IV. Angelic IRQL Checks

This section starts by describing the concept of IRQLs in Windows device drivers, followed by the design of the IrqlCheck angelic rule and our experience with it on internal drivers.

A. IRQL

IRQL (Interrupt Request Level) is a number, ranging from 0 to 31 on x86, which is used to assign priorities to interrupts. An IRQL value is associated with each CPU processor of a system as well as incoming interrupt requests. If a processor is currently at an IRQL value v_1 and an interrupt arrives at level v_2, then the interrupt waits if $v_2 \leq v_1$. Otherwise, the processor's current task is interrupted, its IRQL is raised to v_2 and it starts processing the interrupt.

Some of the important IRQL levels are `PASSIVE_LEVEL` (0), `APC_LEVEL` (1) and `DISPATCH_LEVEL` (2). User-level threads and most kernel-mode operations execute at `PASSIVE_LEVEL`. Since this is the lowest level, all interrupts are accepted at this level. Asynchronous procedure calls (APCs) and the page fault handler execute at `APC_LEVEL`. The Windows thread scheduler and deferred procedure calls (DPCs) execute at `DISPATCH_LEVEL`. When executing at

[6]https://github.com/boogie-org/corral/blob/master/AddOns/PropInst/PropInst/ExampleProperties/doubleFetch.avp

this level, a thread cannot be pre-empted by other threads because an interrupt request from the thread scheduler gets masked (but higher-level interrupts can still be scheduled as they arrive).

Windows provides kernel routines to manipulate the IRQL value of a processor. Driver developers use these routines to control which interrupts should be masked during the execution of the driver. However, the developer must use these routines carefully. For instance, code running at `APC_LEVEL` should not access pageable memory due to the possibility of a page fault, which cannot be served at `APC_LEVEL`. Running at `DISPATCH_LEVEL` further rules out context-switching to other threads. Thus, a thread must not wait on any synchronization objects at this level. Furthermore, the amount of time spent at `DISPATCH_LEVEL` should be limited to a minimum to keep the system responsive. In order to guard against these errors, Windows restricts invoking certain kernel routine at an unacceptable IRQL value. Doing so can cause a kernel panic at runtime. The IRQL requirements of each kernel routine are very clearly specified in the MSDN documentation. For instance, expensive safe-string routines like `RtlEqualString`[7] should only be invoked at `PASSIVE_LEVEL`, etc. It is important to weed out incorrect IRQL violations statically.

B. AV rules

Development of the AV property specification for checking correct IRQL usage required modest effort; a majority of it was completed in one person month. The IRQL requirements of each kernel routine was already well documented. Often, the effort was simply to codify the documentation. For instance, the `KeAcquireSpinLock`[8] routine requires that current IRQL be less than `DISPATCH_LEVEL`. It then raises the IRQL to `DISPATCH_LEVEL` and stashes the old IRQL value in the pointer argument supplied to it. Calls to this routine are instrumented as shown below.

```
procedure KeAcquireSpinLock(x0: int, x1: int);
{
   assert irql <= 2;
   Mem[x1] := irql;
   irql := 2;
}
```

This uses a single global variable `irql` that records the current IRQL value of the processor. Note that SDV performs sequential verification only; correspondingly, we only need to track the IRQL of a single processor.

Any behavior that was unrelated to IRQLs, e.g., actually acquiring a lock, was left unspecified, limiting the amount of work that was required to design the AV property file. The entire property specification consists of 476 such rules. Out of these, only 65 routines actually change the IRQL value, whereas the rest simply assert a precondition. Each rule was at most 4 lines of instrumentation.

[7]https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/ntddk/nf-ntddk-rtlequalstring

[8]https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-keacquirespinlock

```
int DriverRoutine(...)
{
   if(...) // branch B
   {
      // Requires irql <= 1, sets irql to 1
      ExAcquireFastMutex(...);
      // Requires irql == 0
      RtlEqualString(...);
   }
   else
   {
      // Requires irql == 2
      KeTryToAcquireSpinLockAtDpcLevel(...)
   }
   ...
}
```

Fig. 5. Program snippet illustrating AV for checking correct IRQL usage.

The AV vocabulary is set to arbitrary arithmetic constraints (equality, disequality and comparisons) over the variable `irql` and any constant. This vocabulary is different from the vocabulary used for memory safety rules, where we do not permit equality predicates over the pointers. For instance, for NULLCHECK, the vocabulary only allows disequality constraints to model non-aliasing and non-nullness. This illustrates the flexibility of the AV framework to adapt to new classes of rules with relative easy.

We illustrate the behavior of AV for checking correct IRQL usage using the example shown in Figure 5. This program shows a simple driver routine that calls three different kernel routines. The requirements of each of the kernel routines are shown in comments, along with any side-effects they have on changing the IRQL value. This information is instrumented into the program by property instrumentation tool.

We describe a sample run of AV on `DriverRoutine`. AV will start analysis on this routine with an unconstrained initial value for `irql`, because of which it will detect a possible failure of the assertion $\texttt{irql} \leq 1$ at the call to `ExAcquireFastMutex`. AV will suppress this failure by installing an angelic precondition on `DriverRoutine`, namely that $\texttt{irql} \leq 1$. Next, AV restarts the analysis with this new pre-condition. In this case, it will report another possible failure: the assert $\texttt{irql} == 0$ fails on the call to `RtlEqualString`. This failure has no dependence on the initial value of `irql` (because `ExAcquireFastMutex` always sets `irql` to 1); thus, it cannot be suppressed by AV and it will correspondingly show the violation to the user as a single trace of execution of `DriverRoutine`. This kind of analysis also has a pleasant side-effect: the user can inspect this trace in isolation from the callers of `DriverRoutine` because the initial value of `irql` is immaterial.

C. Inconsistency Violations

There is a second class of violations that AV can report that we call *inconsistency violations*. Such a violation consists of multiple (2 or more) traces. Consider the assertion at the call to `KeTryToAcquireSpinLockAtDpcLevel`. AV will try to suppress this violation as well by installing the precondition $\texttt{irql} == 2$ to `DriverRoutine`. However, this precondition *conflicts* with the previous precondition $\texttt{irql} \leq 1$, i.e., their conjunction is *unsatisfiable* (and therefore rejected by the permissiveness check in Figure 2). AV detects this conflict and shows a violation to the user consisting of two traces: the first trace is the one that ends in a call to `ExAcquireFastMutex` and the second one ends in a call to `KeTryToAcquireSpinLockAtDpcLevel`. This violation tells the user that at least one of the calls (maybe, both) is potentially buggy.

Note that it is possible for AV to report false inconsistency violations. For instance, if the branch `B` in the code had a condition dependent on some flags passed to `DriverRoutine` that were set by its callers to indicate different IRQL levels, then it is possible for `DriverRoutine` to be correct. AV only includes inferring preconditions on the `irql` variable, so it does not pick up the branch condition `B`. This allows AV to scale at the cost of some precision. In our experiments, however, AV did not report any false positives because of this reason.

D. Evaluation

During the development of the IRQLCHECK rule, we conducted an initial set of experiments on internal driver code. We picked 797 driver modules that each had some usage of IRQL levels, i.e., ones that called at least one kernel routine that manipulated the IRQL value. The average running time of AV on these modules was 180 seconds. A distribution of the AV running times, shown in Figure 6, indicates that a majority of modules take little time, however there are a few that take much longer.

AV reported a total of 29 violations in these modules. Manual inspection revealed that 26 of these were true violations; of these 18 were single-trace defects and the rest 8 were inconsistency violations consisting of two traces each. There were 3 false positives. In two of these, AV reported a violation assuming an initial IRQL value of -1. This is not possible. We fixed it by constraining the `irql` variable to always be between 0 and 31 and the corresponding false positives went away. One false positive was due to imprecision in aliasing where the value stored under a global variable was overwritten by a write through an unconstrained pointer. The code did not contain any evidence that the pointer could alias the global. This is currently a limitation of AV but happens very rarely; usually the pointer analysis is good at ruling out such possibilities.

V. RELATED WORK

Memory safety of C/C++ applications has naturally received a lot of attention, both in academia and industry, because of their security implications. Microsoft has invested in tools such as Esp [26], Espx [27] and Prefix [28] that have all targeted the Windows Operating System. As opposed to SDV, these tools are not shipped externally. They are used in-house and for that reason they have been heavily tuned for internal Windows code through the use of custom analyses (often a form of pointer analysis), annotations (SAL [29]) or extensive models. Such investment makes the tools expensive to maintain (dependence on annotations/models also implies a constant maintenance struggle) or extend to new properties (which requires a new custom analysis).

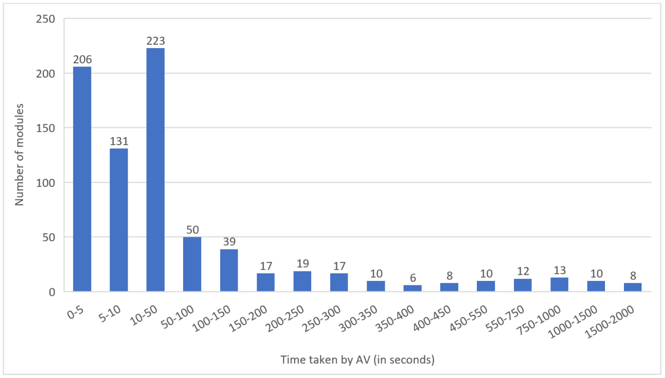

Fig. 6. Histogram of AV running times on multiple Windows modules.

Facebook supports the open-source tool Infer [30]. Infer performs a bottom-up pointer-based analysis of C/C++ programs (even Java) looking for null-safety, use-after-free violations, etc. Infer is designed to be incredibly fast so that developers get immediate feedback as they make code changes. SDV is much more heavy-weight with the use of its SMT-based engine so it cannot provide immediate feedback. Instead, SDV finds its place in a certification process for drivers where it has more time to perform the analysis. On the other hand, Infer is not as readily extensible as SDV for new checks as it requires the creation of a new abstract domain for summarizing behaviors relevant to the property of interest. Besides, the presence of overapproximate summaries can lead to false alarms even for closed programs, especially when summaries need to capture complex arithmetic conditions. Unfortunately, we cannot perform a direct comparison with Infer on the common rules, as Infer cannot be integrated into the build environment for these Windows drivers that use the Microsoft C/C++ compiler toolchain.

The angelic checks in SDV has two key contributions over the tools mentioned above. First, the use of a precise SMT-based backend allows AV-SDV to be easily tuned to support multiple different rules, simply as a new AVP instrumentation file. For instance, IRQLCHECK is not a pointer-based rule; instead it requires arithmetic reasoning of the IRQL value. Yet, AV-SDV supports it without any changes to the tool flow. Second, AV can tolerate imprecise models, thus considerably reducing the maintenance effort.

The core idea of angelic verification is related to research on *abduction* [31] and maximal specification inference [32]. These techniques use novel yet expensive quantifier elimination algorithms to find permissive specifications on the environment. Further these techniques can be used to infer loop invariants for unbounded executions. The main differences lie in the use of AV in SDV to detect high quality bugs instead of finding the maximally permissive or inductive specifications. Although this may result in AV failing to infer a permissive specification even when it exists, AV's lightweight predicate abstraction allows us to scale to modules with hundreds of thousands of lines of code in the presence of a heap. Finally, the idea of starting exploration from functions other than entrypoints is also explored in recent scalable pointer analysis approaches [33], [34].

VI. Conclusion

In this paper, we described our experience integrating angelic checking with the Static Driver Verifier tool, over a period of several years. We described the limitations of SDV for checking unsupported drivers as well as memory safety properties before this work, and provide evidence that the angelic checks provide a cost-effective solution to finding high-quality defects in drivers with very low upfront investment. For future work, we are currently working on: (a) making AV more scalable by pruning state space already explored from transitive callees, and (b) providing support for writing other security critical angelic checks that require *taint* tracking through values in the heap.

References

[1] A. Das, S. K. Lahiri, A. Lal, and Y. Li, "Angelic verification: Precise verification modulo unknowns," in *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, vol. 9206. Springer, 2015, pp. 324–342.

[2] T. Ball, V. Levin, and S. K. Rajamani, "A decade of software model checking with slam," *Communications of the ACM*, vol. 54, no. 7, pp. 68–76, 2011.

[3] Microsoft, "Static driver verifier," https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/static-driver-verifier, 2012.

[4] ——, "Programming reference for windows device driver interface (ddi)," https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi, 2019.

[5] "Windows hardware lab kit," https://docs.microsoft.com/en-us/windows-hardware/test/hlk/, 2018.

[6] T. Ball, E. Bounimova, V. Levin, R. Kumar, and J. Lichtenberg, "The Static Driver Verifier research platform," in *Computer Aided Verification*. Springer, 2010, pp. 119–122.

[7] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of C programs," in *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, M. Burke and M. L. Soffa, Eds. ACM, 2001, pp. 203–213.

[8] T. Ball, E. Bounimova, R. Kumar, and V. Levin, "SLAM2: static driver verification with under 4% false alarms," in *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, R. Bloem and N. Sharygina, Eds. IEEE, 2010, pp. 35–42.

[9] A. V. Nori, S. K. Rajamani, S. Tetali, and A. V. Thakur, "The yogiproject: Software property checking via static analysis and testing," in *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, 2009, pp. 178–181.

[10] A. Lal, S. Qadeer, and S. K. Lahiri, "A solver for reachability modulo theories," in *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, vol. 7358. Springer, 2012, pp. 427–443.

[11] A. Lal and S. Qadeer, "Powering the static driver verifier using corral," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, 2014, pp. 202–212.

[12] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[13] "Microsoft: 70 percent of all security bugs are memory safety issues," https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues, 2019.

[14] S. Blackshear and S. K. Lahiri, "Almost-correct specifications: a modular semantic framework for assigning confidence to warnings," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. ACM, 2013, pp. 209–218.

[15] M. Barnett, K. R. M. Leino, M. Moskal, and W. Schulte, "Boogie: An intermediate verification language," 2009, https://github.com/boogie-org/boogie/.

[16] "Microsoft: The case of the kernel mode double-fetch," https://msrc-blog.microsoft.com/tag/double-fetch/, 2008.

[17] K. R. M. Leino, "Efficient weakest preconditions," *Inf. Process. Lett.*, vol. 93, no. 6, pp. 281–288, 2005.

[18] J. Condit, B. Hackett, S. K. Lahiri, and S. Qadeer, "Unifying type checking and property checking for low-level code," in *ACM SIGPLAN Notices*, vol. 44, no. 1. ACM, 2009, pp. 302–314.

[19] "Openssl tls heartbeat extension read overflow discloses sensitive information," https://www.kb.cert.org/vuls/id/720951/, 2014.

[20] "Cwe-367: Time-of-check time-of-use (toctou) race condition," https://cwe.mitre.org/data/definitions/367.html, 2008.

[21] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim, "Precise and scalable detection of double-fetch bugs in OS kernels," in *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 2018, pp. 661–678.

[22] P. Wang, J. Krinke, K. Lu, G. Li, and S. Dodier-Lazaro, "How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel," in *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, 2017, pp. 1–16.

[23] L. O. Andersen, "Program analysis and specialization for the C programming language," Ph.D. dissertation, DIKU, University of Copenhagen, May 1994.

[24] A. Das and A. Lal, "Precise null-pointer analysis using global value numbering," in *Automated Technology for Verification and Analysis (ATVA)*, July 2017.

[25] "Cwe-416: Use after free," https://cwe.mitre.org/data/definitions/416.html, 2008.

[26] M. Das, S. Lerner, and M. Seigle, "ESP: path-sensitive program verification in polynomial time," in *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, J. Knoop and L. J. Hendren, Eds. ACM, 2002, pp. 57–68.

[27] B. Hackett, M. Das, D. Wang, and Z. Yang, "Modular checking for buffer overflows in the large," in *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, L. J. Osterweil, H. D. Rombach, and M. L. Soffa, Eds. ACM, 2006, pp. 232–241.

[28] W. R. Bush, J. D. Pincus, and D. J. Sielaff, "A static analyzer for finding dynamic programming errors," *Softw. Pract. Exp.*, vol. 30, no. 7, pp. 775–802, 2000.

[29] "Using sal annotations to reduce c/c++ code defects," https://docs.microsoft.com/en-us/cpp/code-quality/using-sal-annotations-to-reduce-c-cpp-code-defects?view=vs-2019, 2016.

[30] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang, "Compositional shape analysis by means of bi-abduction," in *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Z. Shao and B. C. Pierce, Eds. ACM, 2009, pp. 289–300.

[31] I. Dillig, T. Dillig, and A. Aiken, "Automated error diagnosis using abductive inference," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, J. Vitek, H. Lin, and F. Tip, Eds. ACM, 2012, pp. 181–192.

[32] A. Albarghouthi, I. Dillig, and A. Gurfinkel, "Maximal specification synthesis," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, R. Bodík and R. Majumdar, Eds. ACM, 2016, pp. 789–801.

[33] Q. Shi, X. Xiao, R. Wu, J. Zhou, G. Fan, and C. Zhang, “Pinpoint: Fast and precise sparse value flow analysis for million lines of code,” *SIGPLAN Not.*, vol. 53, no. 4, 2018.

[34] H. Yan, Y. Sui, S. Chen, and J. Xue, “Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities,” in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE 18. Association for Computing Machinery, 2018.