Powering the Static Driver Verifier using Corral

Akash Lal
Microsoft Research, India
akashl@microsoft.com

Shaz Qadeer
Microsoft Research, USA
qadeer@microsoft.com

ABSTRACT

The application of software-verification technology towards building realistic bug-finding tools requires working through several precision-scalability tradeoffs. For instance, a critical aspect while dealing with C programs is to formally define the treatment of pointers and the heap. A machine-level modeling is often intractable, whereas one that leverages high-level information (such as types) can be inaccurate. Another tradeoff is modeling integer arithmetic. Ideally, all arithmetic should be performed over bitvector representations whereas the current practice in most tools is to use mathematical integers for scalability. A third tradeoff, in the context of bounded program exploration, is to choose a bound that ensures high coverage without overwhelming the analysis.

This paper works through these three tradeoffs when we applied Corral, an SMT-based verifier, inside Microsoft's Static Driver Verifier (SDV). Our decisions were guided by experimentation on a large set of drivers; the total verification time exceeded well over a month. We justify that each of our decisions were crucial in getting value out of Corral and led to Corral being accepted as the engine that powers SDV in the Windows 8.1 release, replacing the SLAM engine that had been used inside SDV for the past decade.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Testing, Verification

Keywords

Software Verification, SMT, Device Drivers, Bitvector Reasoning, Language Semantics, Loop Coverage

FSE'14, November 16–21, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11...$15.00
http://dx.doi.org/10.1145/2635868.2635894

1. INTRODUCTION

Work on software verification usually focusses on the techniques and algorithms behind the verifiers and their evaluation in a controlled environment. However, when the verifier is applied in practice, it is exposed to several sources of complexity and imprecision, at which point it is important to consider the end-to-end verification system instead of just the exploration algorithms.

We experienced the challenges of a production environment when we applied the Corral verifier inside the Static Driver Verifier (SDV) [16]. SDV is a commercial offering by Microsoft that ships with the Windows Driver Development Kit. Its purpose is to help driver developers find defects earlier in the development cycle and improve the reliability of Windows device drivers. Internally, SDV has used SLAM [1] for statically exploring behaviors of programs. The SDV/SLAM system is one of the major success stories of verification technology [2].

Corral accepts programs in an intermediate verification language called Boogie [3]. In addition to the usual answers of "verified" or "bug found", Corral can also give up when it hits a user-supplied bound on the number of loop iterations (and procedural recursion) to be explored. The original Corral publication [14] showed promise in out-performing SLAM. However, when we applied SDV-Corral combination in a production environment, the number of false defects and missed defects were both at an unacceptable level.

One challenge was defining a memory model for C, i.e., a formal treatment of pointers and the heap in C programs. A bit-precise memory model of C is often intractable. The common trend in many verification tools (such as SLAM [1], SMACK [17], CBMC [5, 7]) is to use a pointer analysis for disambiguating pointer dereferences to different memory locations. This has the disadvantage that the semantics of the pointer analysis get imposed on the subsequent verification. For instance, most pointer analyses assume that environment pointers (pointers allocated outside the scope of the program under test) never alias. This was not a reasonable assumption in our setting. In fact, we were able to find many more defects than SDV/SLAM simply by relaxing this assumption.

One option is to design the memory model based on a pointer analysis that will agree with the verification semantics. However, we do not attempt to go that route. Instead we present a simple pointer disambiguation based on syntactic rules, leaving the semantic heavy lifting to the verifier. Our memory model does not force environment pointers to be distinct.

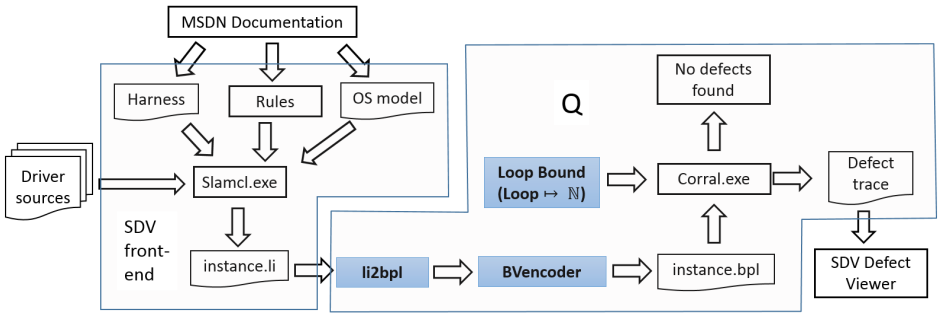

Figure 1: Workflow of the SDV/Q system. This paper's contributions are about the three colored boxes.

The second challenge was modeling of the `int` type in C. On a 32-bit machine (which is assumed to be the case throughout this paper, for simplicity), an `int` is really a 32-bit value. Arithmetic operations such as addition and subtraction are really bitvector operations implemented by circuits inside the hardware. While bitvector-level reasoning is possible—most SMT solvers support a bitvector theory—it is often more expensive than reasoning over mathematical integers. Consistently using bitvector operations in our setting turned out to be too expensive. Instead, we present a novel type-constraint-based analysis that focuses the precision of bitvector reasoning to only where it is needed, whereas the rest of the program computes over mathematical integers. Our experiments validate that we were as precise as full-bitvector reasoning while being almost as efficient as full-integer reasoning.

The third challenge was to ensure coverage and avoid missing defects when present. When Corral is unable to find a proof or defect even after fully exploring a bounded set of behaviors, it terminates search with the verdict "no bugs found". Corral accepts a user-defined bound on the number of loop iterations, as is common in many bounded verifiers. Typically, this bound is chosen upfront. For instance, in the Software Verification Competition SVCOMP [4], many bounded-exploration tools chose a fixed bound of 10—perhaps because it was sufficient to cross all loops in the SVCOMP benchmarks—and this bound is applied to all loops in the program. However, most programs in our test suite had long-running loops. Choosing a bound large enough to cross such loops, and applying it to all loops in the program turned out to be too expensive. We instead apply a different bound to each loop based on a lightweight analysis of the program. Interestingly, in our experiments we found that SLAM, even though it does not do bounded exploration, would time out when the defect required crossing a long-running loop, indicating that our technique is more widely applicable than for bounded-reachability verifiers.

The integration of Corral with the improvements described in this paper is called Q. We show that Q retained much of the speedup that Corral provided in a previous experiment [14], but now it also finds more defects and reports fewer false defects than SLAM. Our evaluation convinced the SDV product team to adopt Q in place of SLAM. The contributions of this paper can be summarized as follows:

1. We present a new memory model for C by defining a translation to Boogie (Section 3).

2. We present a novel type-constraint-based analysis that selectively decides where bitvector reasoning is important (Section 4).

3. We give heuristics that improve coverage across long-running loops (Section 5).

4. We carried out extensive experiments to validate that all of our techniques were necessary for Q to outperform SLAM (Section 6). Our test suite consists of over 5000 verification checks on real drivers that cumulatively take more than 350 hours to execute sequentially.

2. OVERVIEW OF Q

The operation of SDV integrated with Q is shown in Fig. 1. SDV comes packaged with a formal description of a driver's environment and properties in the form of *harnesses*, *OS models*, and *rules*. The harness is like a *main* method that describes how a driver can be invoked. The OS models are stubs for each external call that the driver can make to the kernel, and the rules are properties that the drivers must satisfy [15]. SDV takes the source code of a driver as input and compiles it (using a compiler called `slamcl`) to produce a single file `instance.li`, where the driver is closed using the harness and stubs, and the rule is instrumented as assertions in the program. The format of this file (called `li`) is a simplified C syntax. In this paper, we do not distinguish between `li` and C. This `instance` file is called a verification instance. Each rule results in a different verification instance, thus, a driver produces as many verification instances as the number of rules.

The responsibility of the engine is to find an execution of `instance.li` that ends in an assertion violation. Q works by first compiling C to Boogie (extension `bpl`). The utility `li2bpl` is responsible for encoding the program's C semantics in the more logical language Boogie. The output of `li2bpl` is fed to another utility called `BVencoder` that performs a Boogie-to-Boogie transformation and decides where to introduce bitvector precision. The resulting Boogie file is fed to Corral to find assertion violations. In addition to the Boogie file, Corral also accepts loop bounds as input. This is supplied as a map R from loops to positive integers. Corral then covers at least all executions in which a loop L executes for $R(L)$ number of iterations. Corral additionally accepts a bound on the number of recursive calls (not shown in the figure). However, recursion is not common in drivers and we ignore discussing about it in this paper.

The initial version of SDV/Corral [14] used HAVOC [8] for `li2bpl`, did not support bitvector operations (i.e., over-approximated them as uninterpreted functions), and used a consistent small loop bound for all loops. While the raw performance of this tool compared favorably to SDV/SLAM,

its accuracy in terms of false negatives (missed defects) and false positives (false defects) was far from satisfactory. A more detailed discussion can be found in Section 6.

The rest of the paper is organized as follows. In Section 3, we improve on HAVOC's memory model (in `li2bpl`) by adding support for common programming idioms found in low-level systems code. In Section 4, we describe the design of `BVencoder`. In Section 5, we show how to compute relevant loop bounds. In Section 6, we empirically show why each of these techniques were necessary for Q to outperform SLAM. Section 7 discusses related work and Section 8 concludes.

3. TRANSLATING C TO BOOGIE

Fig. 2 shows a simplified C syntax for expressions and commands. We introduce a `bool` type in C for convenience, and assume that a pre-processing step rewrites expressions involved in a Boolean decision, say `if`(e), to `if`$(e$`!=0`$)$. Further, we do not allow stack-allocated structures. Instead, they must be allocated on the heap. Constants are represented as 32-bit bitvectors (*bv32*). Pointer dereferencing, address-of operator (&) and field indexing is standard. Expressions can be composed of operators. We allow the usual Boolean and arithmetic operators as well as bitwise-and (`&`), bitwise-or (`|`) and bitwise-negation ($\sim$). We use $\otimes$ to represent other non-linear operators in C, such as multiplication, division, bit-shifting, etc. Commands can include assignments as well as **assume** and **assert** statements. We leave out the treatment of control flow and other commands such as procedure calls; their translation is straightforward. Other types like `float` and `union` are either modeled using existing types, or left uninterpreted. Array indexing is modeled using pointer arithmetic. We deliberately do not discuss issues of dealing with the entire C syntax due to space constraints.

Boogie is an imperative language with the usual control-flow constructs of C, however, it does not have a notion of a heap or pointers. The semantics of Boogie is based in logic. The operational semantics of the subset of Boogie that we consider here can be encoded in SMT. The base constructs of Boogie come from standard SMT theories, such as linear arithmetic, bit-vector operations, uninterpreted functions and the theory of arrays.

A Boogie program can only have a finite number of global variables, as well as a finite number of local variables per procedure. Its expression and command syntax is shown in Fig. 3. It has three basic types: `bool` is the usual Boolean type, `int` represents mathematical integers and `bv32` represents 32-bit C integers. Boogie also supports map types. For instance, a variable of type [`int`]`int` represents an (unbounded) map from `int` to `int`. Such map-type variables are crucial for encoding the (unbounded) heap of C programs.

For a map m, $select(m, e)$ indexes the map at location e, and $update(m, e_1, e_2)$ is a map that is identical to m except that its value at location e_1 is e_2. We sometimes shorthand $select(m, e)$ as $m[e]$, and the assignment $m := update(m, e_1, e_2)$ as $m[e_1] := e_2$.

For each C operator `op`, other than the Boolean operators (`&&`, `||`, `!`) and equality (`==`), we assume the existence of two operators $bv_{\texttt{op}}$ and $int_{\texttt{op}}$ in Boogie, where the in and out parameters of $bv_{\texttt{op}}$ have type `bv32`, and the in and out parameters of $int_{\texttt{op}}$ have type `int`. For the purpose of analysis, only some of these Boogie operators can be interpreted precisely by the underlying SMT solver (in our case, Z3 [9]). Others are treated as uninterpreted functions. For instance,

Types	τ	::=	`bool` | `int` | `void*` | τ`*` | `struct S{`$\tau_1\ f_1, \cdots \tau_n\ f_n$`}*`
Variables	x	$\in$	`Var`
Declarations	d	::=	$\tau\ \ x$
Constants	c	$\in$	$\{true, false\} \cup bv32$
L-expressions	l	::=	$x \mid *e \mid l.f$
Expressions	e	::=	$c \mid x \mid \&l \mid l \mid$ $\texttt{op}_2(e_1, e_2) \mid \texttt{op}_1(e) \mid (\tau)e$
Operators	$\texttt{op}_2$	::=	`&&` | `||` | `==` | + | − | $\otimes$ | `&` | `|` | `>`
	$\texttt{op}_1$	::=	`!` | $\sim$
Commands	cmd	::=	$l = e \mid$ **assume** $e \mid$ **assert** e

Figure 2: C expression and command language

Types	δ_{base}	::=	`bool` | `int` | `bv32`
	δ	::=	$\delta_{\text{base}} \mid [\delta_{\text{base}}]\delta_{\text{base}}$
Variables	x	$\in$	`Var`
Declarations	d	::=	$x : \delta$
Constants	c	$\in$	$\{true, false\} \cup \mathbb{Z} \cup bv32$
Expressions	e	::=	$c \mid x \mid op_2(e_1, e_2) \mid op_1(e) \mid$ $select(x, e) \mid update(x, e_1, e_2)$
Operators	op_2	::=	$\wedge \mid \vee \mid == \mid bv_{\texttt{op}_2} \mid int_{\texttt{op}_2}$
	op_1	::=	$\neg \mid bv_{\texttt{op}_1} \mid int_{\texttt{op}_1}$
Commands	cmd	::=	$x := e \mid$ **assume** $e \mid$ **assert** e

Figure 3: Boogie expression and command language

for an arithmetic operator $\texttt{op} \in \{\texttt{+}, \texttt{-}, \texttt{>}\}$, both $int_{\texttt{op}}$ and $bv_{\texttt{op}}$ are interpreted, by the theory of linear arithmetic and the theory of bit-vectors, respectively. Both $int_{\otimes}$ and $bv_{\otimes}$ are uninterpreted. For bitwise-operations $\texttt{op} \in \{\texttt{\&}, \texttt{|}, \sim\}$, $bv_{\texttt{op}}$ is interpreted but $int_{\texttt{op}}$ is uninterpreted. For simplicity, we sometimes use `op` in place of $bv_{\texttt{op}}$ or $int_{\texttt{op}}$ whenever the type of the operator is clear from the context.

The translation of C expressions to Boogie is shown in Fig. 4 using three mutually-recursive routines D (for declarations), E (for expressions) and L (for l-values). The function *typeof* returns the static type of an expression. The function $offset(t, f)$ returns the offset of field f in the structure type t. The function `AddrTaken` takes a C variable as input and returns false if the address of the variable was never taken. The translation is parameterized using the functions `HM` (Heap Map), `OM` (Operator Map) and `TM` (Type Map) that we describe next. The translation of C commands to Boogie commands is straightforward: the command $l = e$ is translated to $E(l) := E(e)$, **assume** e is translated to **assume** $E(e)$, and **assert** e is translated to **assert** $E(e)$.

Fig. 5 shows various options of defining `HM`, `OM` and `TM`. Picking functions subscripted with *int* leads to an `int`-only encoding, whereas picking functions subscripted with *bv* leads to a more precise encoding where C integers are kept as 32-bit bitvectors, and computation is done via bitvector operations. In this section, we only focus on the `int`-encoding; bitvector modeling is discussed in Section 4.

The function `HM` is responsible for encoding memory accesses. Using functions $\texttt{OM}_{int}$, $\texttt{TM}_{int}$ and $\texttt{HM}^{unified}_{\text{int}}$ lead to HAVOC's *unified memory model* (UMM) [6]. The entire heap is represented using one map $\texttt{Mem}_{int}$. (We assume that maps subscripted with *int* have type `[int]int` and maps subscripted with *bv* have type `[bv32]bv32`. The subscripts are dropped when they are clear from the context.)

$$
\begin{array}{lcl}
D(\tau\ x) & = & x : \underline{\texttt{TM}}(\tau) \\
E(c) & = & c \\
E(\&l) & = & L(l) \\
E(x) & = & \begin{cases} \underline{\texttt{HM}}(typeof(x))[x] & \texttt{AddrTaken}(x) \\ x & \text{o/w} \end{cases} \\
E(*e) & = & \underline{\texttt{HM}}(typeof(*e))[E(e)] \\
E(l.f) & = & \underline{\texttt{HM}}(typeof(l), f)[L(l)\ \underline{\texttt{OM}}(+) \\
 & & \quad offset(typeof(l), f)] \\
E(\texttt{op}_1(e)) & = & \underline{\texttt{OM}}(\texttt{op}_1)(E(e)) \\
E(\texttt{op}_2(e_1, e_2)) & = & \underline{\texttt{OM}}(\texttt{op}_2)(E(e_1), E(e_2)) \\
E((\tau)e) & = & E(e) \\
L(x) & = & x \\
L(*e) & = & E(e) \\
L(l.f) & = & L(l)\ \underline{\texttt{OM}}(+) offset(typeof(l), f)
\end{array}
$$

Figure 4: Translation of C to Boogie, parameterized using the underlined functions HM, OM and TM

$$
\begin{array}{lcllcl}
\texttt{TM}_{int}(\texttt{bool}) & = & \texttt{bool} & \texttt{TM}_{bv}(\texttt{bool}) & = & \texttt{bool} \\
\texttt{TM}_{int}(\_) & = & \texttt{int} & \texttt{TM}_{bv}(\_) & = & \texttt{bv32} \\
\texttt{OM}_{int}(\texttt{\&\&}) & = & \wedge & \texttt{OM}_{bv}(\texttt{\&\&}) & = & \wedge \\
\texttt{OM}_{int}(\texttt{||}) & = & \vee & \texttt{OM}_{bv}(\texttt{||}) & = & \vee \\
\texttt{OM}_{int}(\texttt{!}) & = & \neg & \texttt{OM}_{bv}(\texttt{!}) & = & \neg \\
\texttt{OM}_{int}(\texttt{==}) & = & == & \texttt{OM}_{bv}(\texttt{==}) & = & == \\
\texttt{OM}_{int}(\texttt{op}) & = & int_{\texttt{op}} & \texttt{OM}_{bv}(\texttt{op}) & = & bv_{\texttt{op}} \\
\texttt{HM}^{unified}_{int}(\_) & = & \texttt{Mem}_{int} & \texttt{HM}^{unified}_{bv}(\_) & = & \texttt{Mem}_{bv} \\
\texttt{HM}^{unified}_{int}(\_,\_) & = & \texttt{Mem}_{int} & \texttt{HM}^{unified}_{bv}(\_,\_) & = & \texttt{Mem}_{bv} \\
\texttt{HM}^{split}_{int}(t) & = & \texttt{Mem}.t_{int} & \texttt{HM}^{split}_{bv}(t) & = & \texttt{Mem}.t_{bv} \\
\texttt{HM}^{split}_{int}(t,f) & = & \texttt{Mem}.f\_t_{int} & \texttt{HM}^{split}_{bv}(t,f) & = & \texttt{Mem}.f\_t_{bv}
\end{array}
$$

Figure 5: Different translations to Boogie

An alternative is the *split memory model* (SMM) defined by selecting $\texttt{HM}^{split}_{int}$ instead of $\texttt{HM}^{unified}_{int}$. This memory model uses a distinct map for each type and field.[1]

Our standard way of modeling memory allocation is via the procedure `malloc` in Boogie shown in Fig. 6. It returns a strict monotonically increasing integer. This ensures, for instance, that two allocated addresses are distinct.

An example of SMM translation is shown in Fig. 6. `AddrTaken(x)` is false, thus `x` is translated to a scalar variable in Boogie. `AddrTaken(y)` holds, thus, `y` is translated to a Boogie variable that actually represents its address. We use the identifier name `addr_y` in Boogie to highlight this fact.

Suppose `S1` is a structure type with field `f` at offset o_f, and `S2` is a different structure type with field `g` at offset o_g. Then a write to `x->f` translates to a write to `Mem.f_S1[x + `o_f`]` in Boogie. A read of `y->g` translates to a read of `Mem.g_S2[y + `o_g`]`. Thus, irrespective of the values of x, y, o_f, and o_g, an assignment to `x->f` cannot change the value of `y->g` in the translated Boogie program. In other words, an assignment via one field cannot change the value read from another field. This splitting of the memory map statically enforces certain non-aliasing in the program. However, it can also lead to imprecision because even well-behaved C programs can violate this property. Our experiments (Section 6) show that UMM is not scalable whereas SMM results in many false defects.

Refined Memory Model. Our *refined memory model* (RMM) is derived from SMM by selectively merging some of its maps into the same map. In the worst-case, RMM may end up merging all maps into the same map, effectively resulting in UMM. In this sense, the RMM sits between SMM and UMM. It tries to get the scalability of SMM and the precision of UMM.

[1]While writing types inside identifiers, we replace "*" with "P", for example, $\texttt{HM}^{split}_{l}(\texttt{int}*)$ is $\texttt{Mem.PINT}_l$ for $l \in \{bv, int\}$.

```
1  struct S {
2    int *f;
3  }
4
5  void main() {
6   int x, y;
7   S *z;
8   z = malloc(4);
9   z->f = &y;
10  y = x;
11 }
```

```
1  var alloc: int;
2  var Mem.f_S: [int]int;
3  var Mem.INT: [int]int;
4  procedure malloc(size: int)
5      returns (ret: int) {
6   ret := alloc;
7   alloc := alloc + size;
8   return ret;
9  }
10 procedure main() {
11  var x, addr_y, z: int;
12  call addr_y := malloc(4);
13  call z := malloc(4);
14  Mem.f_S[z+0] = addr_y;
15  Mem.INT[addr_y] := x;
16 }
```

Figure 6: Memory allocation and the split memory model over `int`

```
1  typedef struct {
2    int g;
3    int f;
4  } S;
5  void main() {
6     S *x =
7       malloc(sizeof(S));
8     x->f = 1;
9     inc(&x->f);
10    assert(x->f == 2);
11 }
12 void inc(int *a) {
13    (*a)++;
14 }
```

```
1  var Mem.f_S: [int]int;
2  var Mem.INT: [int]int;
3  procedure main() {
4     var x: int;
5     call x := malloc(8);
6     Mem.f_S[x+4] := 1;
7     inc(x+4);
8     assert
9       Mem.f_S[x+4] == 2;
10 }
11 procedure inc(a: int) {
12    Mem.INT[a] :=
13      Mem.INT[a] + 1;
14 }
```

Figure 7: The need for merging fields with types

We merge by defining an equivalence $\equiv$ over the map variables of SMM. Let $[.]_{\equiv}$ be a function that takes a map m as input and returns a unique representative in the equivalence class of m under $\equiv$. We define $\texttt{HM}^{refined}_{l}(t,f) = [\texttt{HM}^{split}_{l}(t,f)]_{\equiv}$ and $\texttt{HM}^{refined}_{l}(t) = [\texttt{HM}^{split}_{l}(t)]_{\equiv}$ for all t and f, and $l \in \{bv, int\}$. The merging equivalence is decided by the following two rules.

Rule 1, taking the address of an expression. Fig. 7 shows a C program and its translation under SMM. The assertion in the C program holds, but not in the Boogie program. The problem is that `inc` updates the map `Mem.INT`, whereas `main` uses `Mem.f_S`. To fix this, whenever the address of a field is taken (`&x->f`), we merge the map for the field with that for the type of the field (`Mem.f_S` $\equiv$ `Mem.INT`). Passing fields by reference is very common in C programs, including drivers. For similar reasons, whenever the address of a variable is taken via the assignment `y = &x` then we say `Mem.`*typeof*`(x)` $\equiv$ `Mem.`*typeof*`(*y)`.

Rule 2, structural subtyping. Fig. 8 shows a C program and its translation under the split-memory model. It illustrates a common idiom in C programs, namely that of structural subtyping. If one type `S1` is a prefix of another type `S2` when they are laid out in memory, then `S2` can be used as a subtype of `S1`. For instance, in Fig. 8, the procedure `inc` is expecting a pointer to `S1`, but `main` is instead passing a pointer to `S2`.

```
1  typedef struct {
2    int g;
3  } S1;
4  typedef struct {
5    int g; int f;
6  } S2;
7  void main() {
8     S2 *x =
9       malloc(sizeof(S2));
10    x->g = 0;
11    inc(x);
12    assert(x->g == 1);
13 }
14 void inc(S1 *a) {
15    a->g = a->g + 1;
16 }
```

```
1  var Mem.g_S1: [int]int;
2  var Mem.f_S2: [int]int;
3  var Mem.g_S2: [int]int;
4
5  procedure main() {
6     var x: int;
7     call x := malloc(8);
8
9     Mem.g_S2[x+0] := 1;
10    inc(x);
11    assert
12      Mem.g_S2[x+0] == 1;
13 }
14 procedure inc(a: int) {
15    Mem.g_S1[a] :=
16      Mem.g_S1[a] + 1;
17 }
```

Figure 8: The need for merging fields with other fields

The assertion in the Boogie program of Fig. 8 can fail because `main` uses `Mem.g_S2` whereas `inc` uses `Mem.g_S1`. Alg. 1 solves this problem by merging structures with their supertypes. The procedure `GetFieldIndex` decorates a field with its type and its offset in the containing structure. We view a structure `S` as an unordered set $[\![S]\!]$ of its fields decorated using `GetFieldIndex`. For example, structure `S2` of Fig. 8 is the set $\{$`g::int::0`, `f::int::4`$\}$. Then, the structure A is structurally a subtype of B if and only if $[\![B]\!] \subseteq [\![A]\!]$.

A naïve algorithm is to simply check $[\![A]\!] \subseteq [\![B]\!]$ for all structures A and B. If n is the total number of structures and m is the maximum number of fields per structure, then this algorithm is $O(n^2m)$ if the set operations are $O(m)$. Alg. 1 does slightly better. It first constructs a dictionary `field2StructSet` that maps a field to all structures that have that field; then the loop on line 7 finds all subtypes of a given structure. Once the subtypes are calculated, we merge field f of structure `S` with field f of structure `S'` whenever `S` is a subtype of `S'` (line 18).

If a field appears in at most p structures, the complexity of Alg. 1 is $O(nmp)$. This is better than the naïve algorithm because p is usually much smaller than n.

Structural subtyping is used commonly in drivers. For instance, drivers have an *extension* object for storing driver-specific state. This extension object is sometimes designed to mirror the driver stack. An example from a `parport` driver is shown in Fig. 9. The declarations of the types make the subtyping intent clear. The `COMMON_EXTENSION` structure declaration is inlined into the declarations of the other structure using a C-preprocessor trick. We also verified that the code of the driver uses this subtyping: methods that expect a `PCOMMON_EXTENSION` are sometimes passed `PFDO_EXTENSION` or `PPDO_EXTENSION`. Our algorithm infers such subtyping relationships. We also note that in our experiments we inferred subtyping relationships that were perhaps never intended or used. Such extra relationships can only have a performance implication, but RMM still retained much of the performance of SMM.

4. BITVECTOR OPERATIONS

The previous section refined the memory model to better model the heap, but it still used `int` as the basic type.

Algorithm 1 Detecting structural sub-typing

Procedure `GetFieldIndex`(f,S)

1: **return** f + "::" + *typeof*(`f`, `S`) + "::" + *offset*(`f`, `S`)

Procedure `StructuralSubtyping`()

```
1: struct2Subtypes = field2StructSet = ∅
2: for all Struct S ∈ AllStructs do
3:    for all Field f in S do
4:       field2StructSet[GetFieldIndex(f, S)].Add(S)
5:    end for
6: end for
7: for all Struct S ∈ AllStructs do
8:    c = AllStructs
9:    for all Field f in S do
10:      if c = ∅ then break
11:      c = c ∩ field2StructSet[GetFieldIndex(f,S)]
12:   end for
13:   struct2Subtypes[S] = c
14: end for
15: for all Struct S ∈ AllStructs do
16:   for all Fields f in S do
17:      for all Structs S' ∈ struct2Subtypes[S] do
18:         Mem.f_S ≡ Mem.f_S'
19:      end for
20:   end for
21: end for
```

```
1  typedef struct _COMMON_EXTENSION {
2    ...
3  } COMMON_EXTENSION, *PCOMMON_EXTENSION;
4
5  typedef struct _FDO_EXTENSION {
6    COMMON_EXTENSION;
7    ...
8  } FDO_EXTENSION, *PFDO_EXTENSION;
9
10 typedef struct _PDO_EXTENSION {
11   COMMON_EXTENSION;
12   ...
13 } PDO_EXTENSION, *PPDO_EXTENSION;
```

Figure 9: Example of structural subtyping

This can be imprecise because all bitwise operations are left uninterpreted. One option is to convert every type to `bv32` by selecting functions in Fig. 5 with subscript *bv*. As we show in Section 6, this option is not scalable. Bitvector reasoning in SMT solvers tends to be much more expensive than integer reasoning.

Our insight is that only some of the computation in a program requires bitvector precision. Most arithmetic (including pointer arithmetic) can be adequately handled using integer operations. We manually inspected the code of some drivers and found that setting and removing of flags as bits of an integer to be an important idiom. Further, these flags maintain important status information about the driver.[2] Such flags are manipulated via bitwise and (`&`), or (`|`), and negation ($\sim$), and are sometimes involved in arithmetic comparisons. The procedure `foo1` in Fig. 12 shows an example of

[2]This idiom is common to almost all drivers. See the description of the `Flags` field of a `DEVICE_OBJECT` structure: `http://msdn.microsoft.com/en-us/library/windows/hardware/ff543147(v=vs.85).aspx`

$$\frac{P \vdash x := e \qquad \Gamma \vdash x : t_x \qquad \Gamma \vdash e : t_e}{t_x = t_e}(\textsc{Assign}) \qquad \frac{\Gamma \vdash \mathit{select}(x, e) : t_1 \qquad \Gamma \vdash x : t_2 \to t_3 \qquad \Gamma \vdash e : t_4}{t_2 = \mathbf{I} \qquad t_1 = t_3}(\textsc{Read})$$

$$\frac{\Gamma \vdash e_1 == e_2 : \mathbf{bool} \qquad \Gamma \vdash e_i : t_i}{t_1 = t_2}(\textsc{Equality}) \qquad \frac{\Gamma \vdash \mathit{int}_{op}(\vec{e_i}) : t \qquad \Gamma \vdash e_i : t_i \qquad op \in \{\&, |, \sim\}}{t_i \leq \mathbf{B} \qquad t \leq t_i}(\textsc{BvOp})$$

$$\frac{\Gamma \vdash \mathit{update}(x, e_1, e_2) : t_1 \to t_2 \qquad \Gamma \vdash x : t_3 \to t_4 \qquad \Gamma \vdash e_1 : t_5 \qquad \Gamma \vdash e_2 : t_6}{t_1 = t_3 = \mathbf{I} \qquad t_2 = t_4 \qquad t_4 = t_6}(\textsc{Write})$$

$$\frac{\Gamma \vdash \mathit{int}_{op}(\vec{e_i}) : t \qquad op \notin \{\&, |, \sim\}}{t = \mathbf{I}}(\textsc{IntOp})$$

Figure 10: Generating type constraints for a Boogie program.

$$\frac{\Gamma \vdash x : \mathbf{B}}{\mathbf{var}\ x : \mathbf{int} \mapsto \mathbf{var}\ x : \mathbf{bv32}}(\textsc{DeclScalar}) \qquad \frac{\Gamma \vdash x : \mathbf{I} \to \mathbf{B}}{\mathbf{var}\ x : \mathbf{[int]int} \mapsto \mathbf{var}\ x : \mathbf{[int]bv32}}(\textsc{DeclMap})$$

$$\frac{\Gamma \vdash e : t}{\mathit{select}(x, e) \mapsto \mathit{select}(x, \mathit{Coerce}(e, t))}(\textsc{Read}) \qquad \frac{\Gamma \vdash e_1 : t}{\mathit{update}(x, e_1, e_2) \mapsto \mathit{update}(x, \mathit{Coerce}(e_1, t), e_2)}(\textsc{Write})$$

$$\frac{\Gamma \vdash \mathit{int}_{op}(\vec{e_i}) : t \qquad \Gamma \vdash e_i : t_i \qquad op \in \{\&, |, \sim\} \qquad \forall i : t_i = \mathbf{B}}{\mathit{int}_{op}(\vec{e_i}) \mapsto \mathit{Coerce}(\mathit{bv}_{op}(\vec{e_i}), t)}(\textsc{BvOpSuccess})$$

$$\frac{\Gamma \vdash \mathit{int}_{op}(\vec{e_i}) : t \qquad \Gamma \vdash e_i : t_i \qquad op \in \{\&, |, \sim\} \qquad \exists i : t_i = \mathbf{I}}{\mathit{int}_{op}(\vec{e_i}) \mapsto \mathit{int}_{op}(\mathit{Coerce}(\vec{e_i}, \vec{t_i}))}(\textsc{BvOpFail}) \qquad \frac{\Gamma \vdash e_i : t_i \qquad op \notin \{\&, |, \sim\}}{\mathit{int}_{op}(\vec{e_i}) \mapsto \mathit{int}_{op}(\mathit{Coerce}(\vec{e_i}, \vec{t_i}))}(\textsc{IntOp})$$

Figure 11: Rewriting the Boogie program

```
1  void foo1(T *x) {
2   ...
3   x->Flags |= F2;
4   ...
5   assert(x->Flags & F3);
6   assert(x->Flags < F4);
7  }
8
9  void foo2(R *y) {
10  g = g + 5;
11  ...
12  y->Data |= F2;
13  ...
14  y->Data = g;
15 }
```

```
1  var Mem.Flags_T: [int]bv32;
2  var Mem.Data_R: [int]int;
3
4  procedure foo1(x: int) {
5    ...
6    Mem.Flags_T[x] := bv|(
7      Mem.Flags_T[x], F2);
8    ...
9    assert bv&(
10     Mem.Flags_T[x], F3)
11     != 0;
12   assert bv2int(
13     Mem.Flags_T[x]) < F4;
14 }
15
16 procedure foo2(y: int) {
17   g := g + 5;
18   ...
19   Mem.Data_R[y] := int|(
20     Mem.Data_R[y], F2);
21   ...
22   Mem.Data_R[y] := g;
23 }
```

Figure 12: Converting types to bitvector

manipulating flags. Each of the `Fi` in the figure are constants (powers of 2).

This section presents `BVencoder`, a Boogie-to-Boogie transformation that takes an `int`-only program and selectively lifts some types to `bv32` when the precision is needed. The result is a mixed `int` and `bv32` typed program, which may additionally use a function `bv2int` that converts a bitvector value to an integer value. Such an operation can be supported precisely; we implement it by a power-of-2 expansion of the bitvector. `BVencoder` does not rely on any conversion from integer to bitvector value.

`BVencoder` works on the following idea. Whenever it sees a bitwise operation, say $\mathit{int}_{\&}(e_1, e_2)$, it tries to change the type of e_1 and e_2 to `bv32` so that the operator can become $\mathit{bv}_{\&}$. However, it does not perform the conversion if, say, e_1 is the result of an arithmetic computation. We now make the analysis more formal.

We introduce three new base types for the purpose of our analysis: $\{\mathbf{N}, \mathbf{B}, \mathbf{I}\}$. `BVencoder`, which takes an `int`-only program, erases the `int` and then re-types the program using these new types. Intuitively, $\mathbf{N}$ stands for "no preference", $\mathbf{B}$ stands for "preferably `bv32`" and $\mathbf{I}$ stands for "must be `int`". These base types satisfy a subtyping relationship $\mathbf{N} > \mathbf{B} > \mathbf{I}$. We say that $t_1 \leq t_2$ if $t_1 = t_2$ or $t_1 < t_2$. For map-types, $t_1 \to t_2 \leq t_3 \to t_4$ only if $t_3 \leq t_1$ and $t_2 \leq t_4$. We make use of a type-coercion method `bv2int` from $\mathbf{B}$ to $\mathbf{I}$.

Every variable and expression in the program is associated with a type variable that can take values from $\{\mathbf{N}, \mathbf{B}, \mathbf{I}\}$ or map-types constructed from these base types. We write $\Gamma \vdash e : t$ when the expression e is associated with type variable t. Type constraints are generated by applying the rules from Fig. 10 on all commands and expressions in the Boogie program. The rule Assign equates the types of right-hand and left-hand sides of an assignment. The rule Equality is similar. Read enforces that the domain types of maps must be $\mathbf{I}$. This is because only pointers can flow in to domains of maps; we wish to keep pointers in the integer domain. Note that Read does not generate a constraint for t_4. Even if the index expression e is typed as $\mathbf{B}$, we can still coerce it to $\mathbf{I}$ using `bv2int`. The rule Write is similar to Read. The rule IntOp forces the result of arithmetic

```
var a,b,c: int;
var x,y,z,w: int;
var m: [int]int;

a := int_&(x, y);
b := int_+(y, z);
c := int_&(b, w);
m := update(m, x, a);
```

$t_a = t_{int_\&(x,y)}$ $\quad t_{int_\&(x,y)} \le t_x$ $\quad t_{int_\&(x,y)} \le t_y$
$t_x \le \mathbf{B}$ $\quad t_y \le \mathbf{B}$
$t_b = t_{int_+(y,z)}$ $\quad t_{int_+(y,z)} = \mathbf{I}$
$t_c = t_{int_\&(b,w)}$ $\quad t_{int_\&(b,w)} \le t_b$ $\quad t_{int_\&(b,w)} \le t_w$
$t_b \le \mathbf{B}$ $\quad t_w \le \mathbf{B}$
$t_m = t_m^{\text{arg}} \to t_m^{\text{res}}$ $\quad t_m^{\text{arg}} = \mathbf{I}$ $\quad t_m^{\text{res}} = t_a$

$\mathbf{N} = t_z$
$\mathbf{B} = t_x = t_y = t_a = t_w = t_m^{\text{res}}$
$\mathbf{I} = t_b = t_c = t_m^{\text{arg}}$

```
var b,c,z: int;
var a,x,y,w: bv32;
var m: [int]bv32;

a := bv_&(x, y);
b := int_+(bv2int(y), z);
c := int_&(b, bv2int(w));
m := update(m, bv2int(x), a);
```

Figure 13: Example showing a Boogie program snippet (left), the type-conversion constraints generated for it (center top), their optimal solution (center bottom), and the converted Boogie program (right).

operations to be **I**. This prevents the need for doing linear arithmetic over bitvector values. The rule BvOp enforces that the arguments of bitwise operators must be at least **B**. If the arguments get typed to be **I** (because, say, they are the result of arithmetic operations) then we cannot convert this operator to its bitvector counterpart. Fig. 13 presents a running example for `BVencoder`.

Once the type constraints are generated, we find the least solution of the constraints. Solving such constraints is standard and very efficient. Note that a solution to the type constraints always exists because forcing all type variables to be **I** is always a valid solution. Once an assignment of the type variables is generated, we re-write the input Boogie program back to use `int` and `bv32` types, using the rewrite rules shown in Fig. 11. Let $Coerce(e, t)$ be e if $t \neq \mathbf{B}$ and `bv2int`(e) otherwise. Thus, the type of $Coerce(e, t)$ is always `int`. Further, for convenience, we say $Coerce((e_1, e_2), (t_1, t_2)) = (Coerce(e_1, t_1), Coerce(e_2, t_2))$.

Fig. 12 shows a C program and its translated Boogie program after `li2bpl` and `BVencoder`. The resulting Boogie program captures precisely the setting and reading of flags in `foo1` but over-approximates in `foo2`.

Our experiments (Section 6) indicate that this strategy of lifting `int` to `bv32` is precise enough for SDV, and in fact improves the running time.

5. IMPROVING LOOP COVERAGE

Consider the program shown in Fig. 14. The `main` procedure has a failing assertion but constructing a path to the assert requires iterating through the loop 28 times. If Corral is given a bound $K < 28$ for this loop, it would not be able to find this defect. Such loops occur commonly in drivers; our experiments show that Q would miss more than 100 defects with a small loop-iteration bound. The loop shown in Fig. 14 is a real example: driver objects have a field `MajorFunction` that is an array of function pointers consisting of the dispatch routines that the driver supports for servicing different operations.[3] It is common for drivers to initialize such arrays by iterating through them.

On the other hand, using a consistently large bound for all loops is not scalable. Our solution is to preferentially assign a larger bound to some loops, while using a default small bound for other loops.

We use two techniques to improve coverage across loops. Let K be the default iteration bound (fixed to 3 in our experiments). Let N be a fixed constant, chosen heuristically to be 50. The two techniques are as follows.

[3] `http://msdn.microsoft.com/en-us/library/windows/hardware/ff551985(v=vs.85).aspx`

```
#define IRP_MJ_MAXIMUM_FUNCTION 27
void main() {
  ...
  for (i = 0; i <= IRP_MJ_MAXIMUM_FUNCTION; i++) {
    DriverObject->MajorFunction[i] = RsPassThrough;
  }
  ...
  assert false;
}
```

Figure 14: Program with a loop

```
procedure foo() {
 ...
 Mem.INT[i] := 0;
 while(Mem.INT[i] < 15) {
   Mem.INT[i] :=
     Mem.INT[i] +1;
   call bar();
 }
 ...
}
procedure bar() {
  Mem.f[...] := e1;
  Mem.g[...] := e2;
}
```

```
var t: int;
procedure foo'() {
 t := 0;
 ...
 Mem.INT[i] := 0;
 while(Mem.INT[i] < 15) {
   t := 1;
   Mem.INT[i] :=
     Mem.INT[i] +1;
   havoc Mem.f;
   havoc Mem.g;
 }
 assert t == 0;
 ...
}
```

Figure 15: Estimating minimum loop iterations

1. *Increasing bound per loop*: if a loop executes for at least m iterations before exiting then we assign it an iteration bound of $K + min(m, N)$.

2. *Abstracting long-running loops*: if a loop requires at least m iterations and $m > N$ then we abstract its $(N+1)^{st}$ iteration to simply havoc the variables (i.e., assign them non-deterministic values) in the loop body and exit the loop.

Ideally, we would have liked to keep $N = \infty$, but in practice some loops require a very large number of iterations. Instead of unrolling these loops, we chose to abstract them instead. This adds the possibility of false defects because we are over-approximating the loops, but we have not seen any such false defects in practice.

We estimate the value of m using Corral itself. Let L be a loop in procedure P. We abstract all procedure calls in P by replacing them with a summary of the called procedure. (The default summary is that the call can arbitrarily modify any variable that it can touch.) Then we run Corral on P with an iteration bound of N. We ask Corral to find a path from the beginning of P to an exit of L. If Corral finds a path with r iterations, then we say $m = r$.

One example is shown in Fig. 15. We estimate minimum iterations for the loop in procedure `foo` by converting it to

Table 1: Various configurations of the Q verification engine.

Engine	Description
Q	Refined memory model, `BVencoder`, loop bound estimation and loop abstraction
Q-RMM	Q but without RMM (uses SMM)
Q-LA	Q but without loop abstraction
Q-LA-LB	Q but without loop-bound estimation and loop abstraction
Q-BV	Q but without `BVencoder` (uses `int`-only encoding)
Q-Base	Split memory model, `int`-only encoding, no loop bound estimation, no loop abstraction
QwIntUMM	Q with unified memory model and `int`-only encoding
QwIntSMM	Q with split memory model and `int`-only encoding
QwAllBv	Q with `bv`-only encoding
QwAllOpsBv	Q where `BVencoder` is applied to all operators
QwLb50	Q without loop-bound estimation but with loop iteration bound of 50 for each loop

the procedure `foo'` shown on the right of the figure. The call to `bar` is abstracted away, and an assertion is added at the end of the loop. Corral will find a path to the assert in `foo'` only after unrolling the loop 15 times, thus, we will set the loop iteration bound for this loop to be $15 + 3 = 18$.

Abstracting procedure calls limits Corral's analysis to a single procedure, making it very efficient. The time spent in estimating bounds for all loops was negligible (mostly less than 5 seconds) compared to the time spent analyzing the original program.

We note that our analysis only provides an estimate of the minimum number of iterations required. It would not be precise, for instance, if the number of iterations is bound to an input parameter of the procedure, and it so happens that all call-sites pass a fixed constant for the parameter. In this case, the value of m inferred would be lower than required and would lead to a loss of coverage. It is possible to perform an interprocedural analysis to infer more precise value of m, however, we did not feel the need for a more complicated analysis in our experiments.

6. EXPERIMENTS

We performed experiments to validate the usefulness of each of our techniques: the refinement memory model (Section 3), `BVencoder` (Section 4) and loop-bound estimation and loop abstraction (Section 5). Let Q refer to the integration of Corral with the various features described in this paper. Different configurations of Q are listed in Table 1.

It is worth noting that the most important result of SDV is a valid defect trace. Finding more defects adds directly to the bottom-line value provided by SDV. Also, reporting a false defect (i.e., one that doesn't reveal a bug in the driver) comes with a high cost because SDV is part of the driver certification process. This is why we put emphasis on the number of missed and false defects in our evaluation.

Test suites. We conducted our experiments over two test suites. The first test suite consists of 4 drivers on which 180 properties were checked. The properties mostly assert that the driver correctly invokes the kernel API [15]. The number of verification instances with non-trivial running time is 616 with 127 of them classified as "buggy". We use this test suite for quick performance testing.

The second test suite is a set of 59 drivers and 180 properties, put together by the SDV team as some of the hardest drivers that SDV comes across in the field. The number of instances with non-trivial running time is 5166, with 1033 of them classified as "buggy". We refer to the first test suite as subITP and the second as NTP (following internally-used names).

Table 2: Comparison of various memory models using int-only encoding on subITP.

	QwIntUMM	QwIntSMM	Q-BV
Time (1000 s)	24.1	14.4	14.9
NURs	242	19	18

The NTP suite is fairly exhaustive; the running time of a single engine on this test suite (excluding compilation) is around 15 days when run sequentially.

The experiments were run in parallel on 4 identical servers. Each of the servers had Intel Xeon CPUs 1.8 GHz, 64 GB RAM and 16 logical processors and ran at most 16 verification instances in parallel (one per core).

Memory models. We first compare the unified, split and refined memory models on subITP test suite. The results are shown in Table 2. NUR stands for "Not Useful Result", which is either a timeout or a spaceout of the verification engine. We consistently use a timeout of 3000 seconds and a spaceout of 2500MB. Also, we report the running time only over instances where the engines reported identical results.

Table 2 clearly shows that the unified memory model has very poor performance, whereas the refined memory model is comparable to the split memory model in terms of the running time. Of course, the issue with split memory model is the inaccuracy in modeling C semantics.

In the evaluation with NTP we also calculated the number of false and missed defects. The results are shown in Table 3; compare the columns for Q and Q-RMM. It is clear that SMM results in many more false defects. Figs. 7 and 8 capture the reasons for these false defects. Moreover, none of the 7 false defects of Q were because of inaccuracies in the refined memory model.

We also note that on an average, RMM produced 14% fewer maps than SMM, with field referencing and structual subtyping contributing almost equally to the merging. The small percentage of map merging is one reason why the performance of RMM was still closer to SMM than UMM.

Bit-vector Analysis. Table 5 shows a comparison of various integer/bitvector encodings under the refined memory model. We also created a version QwAllOpsBv that uses a variant of `BVencoder` that also tries to lift arithmetic to be over bitvector values, but still forces pointer arithmetic and pointer dereferences to be over integers. We use QwAllOpsBv to evaluate if the arithmetic really needs to be over integers or not.

Table 3: Performance of various tools on NTP benchmarks. Speedups are relative to the running time of Q.

	SLAM	Q	Q-BV	Q-RMM	Q-LA	Q-LA-LB	Q-Base
Speedup over Q	0.56	1	0.88	0.98	1.02	1.03	1.1
False defects	10	7	≥67	≥65	8	8	≥108
Missed defects	181	23	23	23	33	109	110
NURs	430	209	237	218	202	204	203

Table 4: Comparison with different loop bounds on subITP.

	Q-BV	Q	QwAllOpsBv	QwAllBv
Time (1000 s)	16.9	16.6	22.2	50.6
NURs	18	18	80	181

Table 5: Comparison of bitvector encodings using the refined memory model on subITP.

	Q	QwLb50
Time (1000 s)	34.0	41.0
NURs	18	43

From Table 5, it is clear that an all-BV encoding is not scalable. In fact, we noticed that as soon as we allowed maps with `bv32` type domain (i.e., maps of type `[bv32]bv32` or `[bv32]int`), the performance of Corral (and Z3) decreased dramatically. This is a topic for separate investigation. Note that `BVencoder` never produces maps of such types. The performance of QwAllOpsBv is much better than QwAllBv, but still does not compare to the performance of Q.

Surprisingly, Q was slightly faster than Q-BV even though Q-BV is purely integer-based. This is perhaps because the less precision of Q-BV forced it to carry out a larger exploration of program behaviors than Q.

The NTP suite reveals that Q-BV results in many false defects (compare columns for Q and Q-BV). None of the 7 false defects of Q were because of less-precise bitvector reasoning, meaning that `BVencoder` was sufficient for dealing with bitvector operations on the benchmarks. The percentage of maps whose type was changed by `BVencoder` was 3.4% on an average. (For QwAllOpsBV, this number was 28.4%).

Loop coverage. Recall that the limit used for loop-bound inference was 50, after which loop abstraction (if enabled) would abstract the loop. Each of Q-LA and Q-LA-LB are strict under-approximations of Q, i.e., they explore strictly less amount of the state-space of a program than Q. First, Table 4 shows that consistently using a bound of 50 is not scalable. The NTP suite shows that Q-LA-LB has a large number of missed defects (109). Many of them (except 33) are found using loop-bound inference. A further 10 were found using loop abstraction. Interestingly, in all of these 10 cases, SLAM timed out, indicating that the loop abstraction heuristic could have benefited SLAM as well. Furthermore, the performance of Q is comparable (within 3%) to the time taken by Q-LA or Q-LA-LB.

Corral vs. SLAM. Table 3 shows that Q was 1.8 times faster than SLAM: for verification instances on which SLAM and Q returned the same (and non-NUR) answer, SLAM took 876000 seconds and Corral took 495000 seconds. Moreover, Q reported only half as many NURs, had fewer false defects and reported 158 more true defects. Thus, Q outperforms SLAM on all metrics. The configuration Q-Base is what was used for experimentation in an earlier Corral paper [14]. While we retained much of the performance improvement offered by Corral, Q performs significantly better on other metrics.

From the 158 new defects found by Q, 88 of them were found because Q's memory model (unlike SLAM) does not assume that environment pointers are distinct. On the flip side, the 7 false defects of Q were because the aliasing that it assumed in the environment was indeed unreasonable in that setting.

To illustrate a real example, consider `DispatchRoutine1` of Fig. 16. The intention is to check if the dispatch routine deletes the device object passed to it (`DO` is a shorthand for `DeviceObject`). Indeed it does; the `Self` field is always expected to point back to the containing device object. However, not all fields (including `Self`) of the device extension (shorthand `DE`) are initialized by the SDV harness (for various reasons outside the scope of this paper). SLAM believes that because `de->Self` is uninitialized, it cannot alias the device object. As a result, it misses this valid defect.

Consider `DispatchRoutine2` in Fig. 16. It checks that the `IRP` sent to the dispatch routine is not completed twice. The two `IRP`s that are completed come from uninitialized fields of the device extension object. It is not immediately obvious if this is a true defect or not. Experts in the SDV team flag this as a false defect because it is expected that the `FlushIrp` and `BlockIrp` are always distinct IRPs. (We have shortened the identifier names from their actual names in the drivers). SLAM does not report this false defect because of its distinctness assumption but Q reports this defect.

The assertion violation in both `DispatchRoutine1` and `DispatchRoutine2` requires aliasing among environment (uninitialized) pointers. However, the fact that one is a true defect while the other is false, purely depends on domain knowledge. Our experiments reveal that we find 88 new defects, and report only 7 extra false defects. This was an interesting side-effect of the change from SLAM to Q; we did not have prior knowledge that SLAM's environment non-aliasing was actually too strong in many cases.

In 65 of the new defects found by Corral, SLAM timed out before returning an answer. The rest ($158 - 88 - 65 = 5$) seem to be because of loss of coverage (or bugs) in SLAM; we did not investigate this further.

Comparison with Yogi. Yogi [11] is another verification engine compatible with SDV. Q outperformed Yogi on all metrics as well, but we avoid going into details due to space constraints. We also note that the version of Q that shipped with SDV includes Yogi as backup engine. Yogi is executed if Corral returns NUR.

Training and Validation. We note that all of our debugging and training of heuristics was performed on the ITP suite of drivers (superset of subITP). The NTP suite was reserved for validation.

Threats to Validity and Limitations. The hardest part of our experiments was classifying defects as true or false. This is a manual effort and very challenging for a large test suite like the NTP. The Quality Assurance team

```
enum {INIT, DELETED};
int t;
DO *global_devobj;

void DispatchRoutine1(DO *devobj) {
  t = INIT;
  global_devobj = devobj;
  DE *de = devobj->DeviceExtension;
  ...
  IoDeleteDevice(de->Self);
  ...
  assert t != DELETED;
}

void IoDeleteDevice(DO *d) {
  if(d == global_devobj)
    t = DELETED;
  ...
}

// Aliasing among environment pointers:
// devobj->DeviceExtension->Self == devobj
```

```
int completed;
IRP *global_irp;

void DispatchRoutine2(DO *devobj, IRP *irp) {
  completed = 0;
  global_irp = irp;
  DE *de = devobj->DeviceExtension;
  ...
  IoCompleteRequest(de->FlushIrp);
  ...
  IoCompleteRequest(de->BlockIrp);
  ...
}
void IoCompleteRequest(IRP *p) {
  if(p == global_irp) {
    assert completed != 1;
    completed = 1;
  }
}
// Aliasing among environment pointers:
// devobj->DeviceExtension->FlushIrp == irp AND
// devobj->DeviceExtension->BlockIrp == irp
```

Figure 16: Examples demonstrating environment aliasing

of SDV put in this manual effort to provide us with the defect classification of SLAM and Q. Classification for other configurations of Q was carried out by us. It is possible that we may have missed classifying some defects as false. However, note that such false defects would only make Q look better.

All of the techniques presented in this paper were inspired by looking at device drivers. While we believe that the fundamentals behind these techniques will generalize to other settings, it is possible that the specific details might not apply to other programs. For instance, the choice of which bitvector operations to support in the `BVencoder` or choosing $N = 50$ before we abstract loops might be specific to drivers. In other settings, these choices may be made differently, but the techniques can still be useful.

7. RELATED WORK

Memory model. The translation of C to Boogie in HAVOC [8], SMACK [17], and now SDV/Q, encodes the heap using one or more map variables. HAVOC uses the split memory model, whereas SMACK uses a pointer analysis for the splitting (and borrows the semantics of the pointer-analysis on environment pointers). Both of these were inadequate in our setting, prompting us to design the refined memory model.

There are other tools that encode C's operational semantics without using maps. For instance, CBMC [7] uses a pointer-analysis to identify the set of all possible targets of a pointer and replaces the dereferences of that pointer with an if-then-else that writes directly to the target locations. In this way, CBMC ends up with a program with only scalar variables. Because of the heavy use of pointer analysis, CBMC has been most successful in the context of embedded systems (that are usually not heap intensive) and doesn't scale as much on the drivers in our test suite.

Bitvector analysis. Our design of `BVencoder` is a novel way of using type-constraint-based analysis to mix integer and bitvector reasoning. Type-constraint-based analyses, and type inference has a rich history of proving program properties. For instance, CQual [10] refines C types with qualifiers, such as *non-null*. Then proving the absence of null dereferences reduces to a type-inference problem.

Loop coverage. Unrolling a loop for a fixed number of times has been the most common approach used with bounded model checkers. There has been recent work to address loss of coverage. In [13], the authors under-approximate a loop using a quantified constraint that captures the mutations performed on the loop, and then replace the loop with this constraint. However, in their case, the target solver was a SAT solver, and quantification of Boolean formulas is still decidable. In Corral, adding quantifiers in the presence of maps and arithmetic leads to undecidability. Nonetheless, it would be interesting to try their approach in our setting.

Work on estimating worst-case complexity of a program [18, 12] also infers bounds on the number of loop iterations. However, that work mostly focusses on finding the maximum number of iterations for a loop in an execution, whereas we are interested in finding the minimum number of iterations before the loop can exit. In other words, worst-case complexity infers the big-O complexity, whereas we want the small-o complexity. In that sense, our heuristic on loop-bound inference is unique.

8. CONCLUSION

This paper gives a thorough walkthrough of the SDV/Q verification system. The act of replacing SLAM with Corral led to many interesting challenges that had to be solved before SDV/Q reached production quality. We used a large set of SDV benchmarks to experiment and learn ways of solving the challenges. Traditionally, the verification community has focussed on the core algorithms while the end-to-end details of the verification system as a whole get ignored. We believe that it is very important to present and understand such details for wider deployment of verification tools.

9. REFERENCES

[1] T. Ball, E. Bounimova, R. Kumar, and V. Levin. SLAM2: Static driver verification with under 4% false alarms. In *Formal Methods in Computer Aided Design*, pages 35–42, 2010.

[2] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with SLAM. *Commun. ACM*, 54(7):68–76, 2011.

[3] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects*, pages 364–387, 2005.

[4] D. Beyer, editor. *1st International Competition on Software Verification, co-located with TACAS 2012, Tallinn, Estonia*, 2012.

[5] CBMC: Bounded Model Checking for ANSI-C. `http://www.cprover.org/cbmc/`.

[6] S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamaric. A reachability predicate for analyzing low-level software. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 19–33, 2007.

[7] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, 2004.

[8] J. Condit, B. Hackett, S. Lahiri, and S. Qadeer. Unifying type checking and property checking for low-level code. In *Principles of Programming Languages*, 2009.

[9] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[10] J. S. Foster. *Type Qualifiers: Lightweight Specifications to Improve Software Quality*. PhD thesis, University of California, Berkeley, Dec. 2002.

[11] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *Principles of Programming Languages*, pages 43–56, 2010.

[12] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *Principles of Programming Languages*, pages 127–139, 2009.

[13] D. Kroening, M. Lewis, and G. Weissenbacher. Under-approximating loops in c programs for fast counterexample detection. In *Computer Aided Verification*, pages 381–396, 2013.

[14] A. Lal, S. Qadeer, and S. Lahiri. Corral: A solver for reachability modulo theories. In *Computer Aided Verification*, 2012.

[15] Microsoft. DDI compliance rules. `http://msdn.microsoft.com/en-us/library/windows/hardware/ff552840(v=vs.85).aspx`.

[16] Microsoft. Static driver verifier. `http://msdn.microsoft.com/en-us/library/windows/hardware/ff552808(v=vs.85).aspx`.

[17] Z. Rakamaric and M. Emmi. SMACK: Static Modular Assertion ChecKer. `http://smackers.github.io/smack`.

[18] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem: overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.