Towards scalable modular checking of user-defined properties

Thomas Ball[1], Brian Hackett[2], Shuvendu K. Lahiri[1]
Shaz Qadeer[1], and Julien Vanegue[1]

[1] Microsoft
[2] Stanford University

Abstract. Theorem-prover based modular checkers have the potential to perform scalable and precise checking of user-defined properties by combining path-sensitive intraprocedural reasoning with user-defined procedure abstractions. However, such tools have seldom been deployed on large software applications of industrial relevance due to the annotation burden required to provide the procedure abstractions.
In this work, we present two case studies of applying a modular checker HAVOC to check properties on large modules in the Microsoft Windows operating system. The first detailed case study describes checking the synchronization protocol of a core Microsoft Windows component with more than 300 thousand lines of code and 1500 procedures. The effort found 45 serious bugs in the component with modest annotation effort and low false alarms; most of these bugs have since been fixed by the developers of the module. The second case study reports preliminary user experience in using the tool for checking security related properties in several Windows components. We describe our experience in using a modular checker to create various property checkers for finding errors in a well-tested applications of this scale, and our design decisions to find them with low false alarms, modest annotation burden and high coverage.

1 Introduction

Developing and maintaining systems software such as operating systems kernels and device drivers is a challenging task. They consist of modules often exceeding several hundred thousand to millions of lines of code written in low-level languages such as C and C++. In many cases, these modules evolve over several decades where the original architects or developers have long ago departed. Such software may become fragile through the accumulation of new features, performance tuning and bug fixes, often done in an ad-hoc manner. Given the astronomical number of paths in any real program, testing can only cover a relatively very small fraction of the paths in a module. Bugs found in the field often occur in these rarely exercised paths.

Static analysis tools provide an attractive alternative to testing by helping find defects without requiring concrete inputs. However, the applicability of completely automatic static tools is limited due to several factors:

– First, most static analysis tools check *generic* properties of code such as buffer overrun, null dereference or absence of data-races. These checkers are not *extensi-*

ble, i.e., they cannot be easily augmented to create a checker for a new user-defined property — testing still remains the only way to check such properties.
- Second, most scalable static analysis tools are based on specific abstract domains or dataflow facts. These tools generate numerous false alarms when the property being checked depends on system-specific invariants that fall outside the scope of the analysis. This happens particularly when the property depends on the heap — even when the property being checked is a generic property as above.
- Finally, more extensible tools (such as those based on predicate abstraction) have scalability problems to large modules because they try to automatically find a proof of the property by searching an unbounded space of proofs. They rely on various automated refinement strategies which are not robust enough to generate all non-trivial invariants for large modules.

Contract-based modular checkers such as ESC/Java [17], Spec# [4], HAVOC [5] and VCC [9] have the potential to perform scalable checking of user-defined properties. These checkers share the following strengths:

1. They provide the operational semantics of the underlying programs irrespective of the property being checked. This is in stark contrast to static analyzers based on data-flow analysis or abstract interpretation, which require defining abstract semantics for each new property.
2. They use a theorem prover to perform precise intraprocedural analysis for loop-free and call-free programs, in the presence of contracts for loop and called procedures.
3. They provide an extensible contract language to specify the properties of interest, and contracts. The use of theorem provers allow rich contracts to be specified, when required, to remove false alarms.
4. Generic interprocedural contract inference techniques (e.g. Houdini [16]) exist to infer contracts to relieve the user from manually annotating the entire module. By allowing the user to provide a restricted space of procedure abstractions (contracts) to search for proofs, the approach allows the user to aid the analysis to find proofs in a scalable fashion.
5. Finally, the presence of contracts provide *incremental* checking across changes to procedures without reanalyzing the entire module, and the contracts can serve as valuable documentation for maintaining these large codebases.

In spite of the potential benefits offered by modular checkers, such tools have been seldom deployed successfully on large software applications of industrial relevance. We believe this is due to the following limitations:

1. The annotation burden for checking a property on such a large code-base can be substantial, and can often be several times the size of the source code. Although contract inference has been proposed to relieve the user burden, previous work in ESC/Java [16, 15] does not allow for inferring user-defined contracts. We provide one particular way for inferring a class of contracts from module invariants [21], but it has not been shown to scale to modules considered in this work.
2. The problem of capturing the side-effect of each procedure and aliasing between pointers can be difficult. Various ownership and encapsulation methodologies have

been proposed [4], but they impose restrictions on the heap manipulation that are often not satisfied by low-level systems code.
3. Finally, there is a lack of good case studies illustrating the feasibility of using such a tool on real-world software to provide value in discovering hard-to-find bugs, with modest investment of user effort.

In this paper, we present a feasibility study of using contract-based modular checkers for cost-effective checking of user-defined properties on large modules of industrial relevance. We first describe our experience with applying the modular checker HAVOC [5, 20] on a core component COMP of the Windows kernel — the name of the module and the code fragments have been modified for proprietary reasons. The code base has more than 300 thousand lines of C code and has evolved over two decades. The module has over 1500 procedures, with some of the procedures being a few thousand lines long — a result of the various feature additions over successive versions. For this component, we specified and checked properties related to the synchronization protocol governing the management of its main heap allocated data structures. The correctness checking of the protocol was decomposed into checking for correct reference counting, proper lock usage, absence of data races and ensuring that objects are not accessed after being reclaimed (teardown race). Verification of these properties required expressing many system-specific intermediate invariants (see Section 2) that are beyond the capabilities of existing static analysis tools. The highlights of the effort that was conducted over a period of two months were:

1. We found 45 bugs in the COMP module that were confirmed by the developers and many of them have been fixed at the time of writing. Most of these bugs appear along error recovery paths indicating the mature and well-tested nature of the code and signifying the ability of modular checkers to detect subtle corner cases.
2. The checking required modest annotation effort of about 250 contracts for specifying the properties and operating system model, 600 contracts for procedure contracts. The contract inference generated around 3000 simple contracts, a bulk of the required annotation effort, to relieve the need for annotating such a large code base. This corresponds to roughly one manual contract per 500 lines of code, or one per 2.5 procedures.
3. The tool currently reports 125 warnings, including the 45 confirmed bugs, when the checker runs on the annotated code base. The extra warnings are violations of intermediate contracts that can be reduced with additional contracts.

Next, we report on preliminary user experience in using the tool for checking security related properties in several other Windows components. Various property checkers have been constructed using HAVOC to check for correct validation of user pointers, and restricted class of exploitable buffer overrun problems. The tool has been deployed on more than 1.3 million lines of code across three or four large components each measuring several hundred thousand lines of code. The effort has yielded around 15 security vulnerabilities that have been already patched.

We describe the challenges faced in using a modular checker for finding errors in well-tested applications of this scale, and our design decisions to find them with low false alarms, modest contract burden and high coverage. Our decisions allowed us to

```
typedef struct _LIST_ENTRY{
  struct _LIST_ENTRY *Flink, *Blink;
} LIST_ENTRY, *PLIST_ENTRY;

typedef struct _NODEA{
  PERESOURCE Resource;
  LIST_ENTRY NodeBQueue;
  ...
} NODEA, *PNODEA;

typedef struct _NODEB{
  PNODEA     ParentA;
  ULONG State;
  LIST_ENTRY NodeALinks;
  ...
} NODEB, *PNODEB;
```

```
#define CONTAINING_RECORD(addr, type, field)\
   ((type *)((PCHAR)(addr) -                \
             (PCHAR)(&((type *)0)->field))) \

//helper macros
#define ENCL_NODEA(x)                       \
    CONTAINING_RECORD(x, NODEA, NodeBQueue) \
#define ENCL_NODEB(x)                       \
    CONTAINING_RECORD(x, NODEB, NodeALinks) \
```

Fig. 1. Data structures and macros used in the example.

achieve an order of magnitude less false alarms compared to previous case studies using modular checkers [16], while working on C modules almost an order more complex than these previous case studies. We believe that the studies also contribute by identifying areas of further research to improve the applicability of these modular checkers in the hands of a user.

2 Overview

In this section, we use the example of checking data-race freedom on the main data structures of COMP to illustrate some of complexities of checking properties of systems software with low-false alarms. In particular, we show that precise checking of even a generic property such as data-race freedom often requires:

- contracts involving pointer arithmetic and aliasing,
- conditional contracts, and
- type invariants to capture aliasing relationships.

Such requirements are clearly beyond the capabilities of existing automated software analysis tools that scale to such large components. This justifies the use of modular checkers that involve the users to decompose the problem using domain-specific knowledge.

We first describe high-level details of the data structure and the synchronization protocol, some procedures manipulating these structures, and finally the contracts to check the absence of data-races.

2.1 Data structures

Figure 1 describes a few types for the heap-allocated data structures in COMP. The type `LIST_ENTRY` is the generic type for (circular) doubly-linked lists in most of Windows source code. It contains two fields `Flink` and `Blink` to obtain the *forward* and *backward* successors of a `LIST_ENTRY` node respectively in a linked list. An object of type `NODEA` contains a list of children objects of type `NODEB` using the field

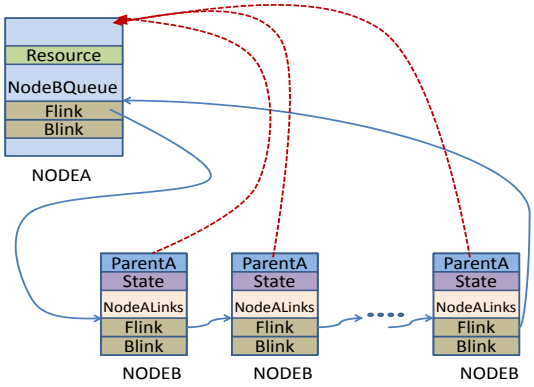

Fig. 2. The list of NODEB children of a NODEA.

NodeBQueue. Figure 2 describes the shape of the children list for any NODEA object. Each child NODEB node also maintains pointers to its *parent* NODEA object with the ParentA field.

The macro CONTAINING_RECORD (defined in Figure 1) takes a pointer addr to an internal field field of a structure of type type and returns the pointer to the enclosing structure by performing pointer arithmetic. The helper macros ENCL_NODEA and ENCL_NODEB uses the CONTAINING_RECORD macro to obtain pointers to enclosing NODEA and NODEB structures respectively, given a pointer to their LIST_ENTRY fields. The CONTAINING_RECORD macro is frequently used and is a major source of pointer arithmetic.

Since these objects can be accessed from multiple threads, one needs a synchronization mechanism to ensure the absence of data-races on the fields of these objects. Each NODEA structure maintains a field Resource, which is a pointer to an ERESOURCE structure that implements a reader-writer lock. The lock not only protects accesses to the fields in the NODEA structure but additionally also protects the fields NodeALinks, ParentA and State in all of its NODEB children.

2.2 Procedures

Figure 3 describes three procedures that manipulate the NODEA and NODEB objects. Contracts are denoted by __requires, __ensures and __loop_inv. ClearChild takes a NODEA object NodeA and clears a mask StateMask from the State field of any NODEB child that has this mask set. It uses the procedure FindChild in a loop to find all the children that have the StateMask set and then clears the mask on the child by calling ClearState. Finally, the procedure FindChild iterates over the children for a NODEA object and returns either the first child that has the mask set, or NULL if no such child exists.

To encode the data-race freedom property on the fields of NODEA and NODEB objects, we introduce assertions that each access (read or write) to a field is guarded by

```
#define __resA(x)         __resource(‘‘NODEA_RES’’,x)
#define __resrA_held(x)   __resA(x) > 0

VOID ClearChild(PNODEA NodeA, ULONG StateMask) {
  AcquireNodeAExcl(NodeA);
  PNODEB NodeB;
  FindChild(NodeA, StateMask, &NodeB);

  __loop_inv(NodeB != NULL ==> NodeB->ParentA == NodeA)
  while (NodeB != NULL) {
     ClearState(NodeB, StateMask);
     FindChild(NodeA, StateMask, &NodeB);
  }
  ReleaseNodeA(NodeA);
}

__requires(__resrA_held(NodeA))
__ensures (*PNodeB != NULL ==> (*PNodeB)->ParentA == NodeA)
VOID FindChild(PNODEA NodeA, ULONG StateMask, PNODEB* PNodeB) {
  PLIST_ENTRY Entry = NodeA->NodeBQueue.Flink;

  __loop_inv(Entry != &NodeA->NodeBQueue ==> ENCL_NODEB(Entry)->ParentA == NodeA)
  while (Entry != &NodeA->NodeBQueue) {
     PNODEB NodeB = ENCL_NODEB(Entry);
     if (NodeB->State & StateMask != 0) {
         *PNodeB = NodeB; return;
     }
     Entry = Entry->FLink;
  }
  *PNodeB = NULL; return;
}

__requires(__resrA_held(NodeB->ParentA))
VOID ClearState(PNODEB NodeB, ULONG StateMask) {
   NodeB->State &= ˜StateMask;
}
```

Fig. 3. Procedures and contracts for data-race freedom.

the `Resource` lock in the appropriate `NODEA` object. The three procedures clearly satisfy data-race freedom since the lock on the `NODEA` object is acquired by a call to `AcquireNodeAExcl` before any of the operations.

2.3 Contracts

Now, let us look at the contracts required by `HAVOC` to verify the absence of the data-race in the program. The procedure `ClearState` has a *precondition* (an assertion inside `__requires`) that the `Resource` field of the `NodeB->ParentA` is held at entry; this ensures that the access to `NodeB->State` is properly protected. The `__resrA_held(x)` macro expands to `__resource`(“`NODEA_RES`”, `x > 0`), which checks the value of a *ghost field* “`NODEA_RES`” inside `x`. The integer valued ghost field “`NODEA_RES`” tracks the state of the re-entrant `Resource` lock in a `NODEA` object — a positive value denotes that the `Resource` is acquired. For brevity, we skip the contracts for `AcquireNodeAExcl` and `ReleaseNodeA`, which increments and decrements the value of the ghost field, respectively.

```
#define FIRST_CHILD(x)   x->NodeBQueue.Flink
#define NEXT_NODE(x)     x->NodeALinks.Flink

__type_invariant(PNODEA x){
    ENCL_NODEA(FIRST_CHILD(x)) != x  ==>
    ENCL_NODEB(FIRST_CHILD(x))->ParentA == x
)

__type_invariant(PNODEB y){
    NEXT_NODE(y) != &(y->ParentA->NodeBQueue) ==>
    y->ParentA == ENCL_NODEB(NEXT_NODE(y))->ParentA
)
```

Fig. 4. Type invariants for NODEA and NODEB types.

The procedure FindChild has a similar precondition on the NodeA parameter. The procedure also has a *postcondition* (an assertion inside __ensures) that captures the child-parent relationship between the out parameters PNodeB and NodeA.

Let us inspect the contracts on ClearChild. We need a *loop invariant* (an assertion inside __loop_inv) to ensure the precondition of ClearState inside the loop. The loop invariant states that NodeB is a child of NodeA when it is not NULL. The postcondition of FindChild ensures that the loop invariant holds at the entry of the loop and also is preserved by an arbitrary iteration of the loop.

Finally, consider the loop invariant in procedure FindChild: the loop invariant is required for both proving the postcondition of the procedure, as well as to prove the absence of a data-race on NodeB->State inside the loop. This loop invariant does not follow directly from the contracts on the procedure and the loop body.

To prove this loop invariant, we specify two *type invariants* for NODEA and NODEB objects using the __type_invariant annotation in Figure 4. The type invariant on any NODEA object x states that if the children list of x is non-empty then the parent field ParentA of the first child points back to x. The type invariant for any NODEB object y states that if the next object in the list is not the head of the circular list, then the next NODEB object in the list has the same parent as y. The two type invariants capture important shape information of the data structures and together imply that all the NODEB objects in the children list of NodeA point to NodeA.

3 Background on HAVOC

In this section, we provide some background on HAVOC, including the contract language, the modular checker and an interprocedural contract inference. In addition to the details of HAVOC described in earlier works [5, 6], we describe the main additions to the tool for this paper. This includes adding support for *resources* and *type invariants* in contracts, and the instrumentation techniques.

3.1 Contracts

Our contracts are similar in spirit to those found in ESC/Java [17] for Java programs, but are designed for verifying systems programs written in C. We provide an overview

of the subset of contracts that are used in this work. Throughout this paper, we use the terms "contracts" and "annotations" interchangeably, although the former is primarily used to express an assertion. More details of the contract language are described in the HAVOC user manual[1].

Procedure contracts and loop invariants. Procedure contracts consist of preconditions, postconditions and modifies clauses. The `__requires` contract specifies a precondition that holds at the entry to a procedure. This assertion is assumed when analyzing the body of the procedure and checked at all call-sites of the procedure. The `__ensures` contract specifies a postcondition that holds at exit from the procedure. The `__modifies` contract specifies a set of locations that are possibly modified by the procedure; it generates a postcondition that *all other* locations in the heap remain unchanged. The postconditions are checked when analyzing the body of the procedure, and assumed at all call-sites for the procedure.

The `__loop_inv` contract specifies a loop invariant — an assertion that holds every time control reaches the head of the loop. The assertion should hold at entry to the loop, and should be preserved across an arbitrary iteration of the loop.

Contract expressions. A novel feature of our contract language is that it allows most call-free and side-effect free C expressions in the assertions. The assertions can refer to user defined macros, thereby allowing complex assertions to be constructed from simpler ones. We allow reference to the return value of a procedure with the `__return` keyword. The postconditions may also refer to the state at the entry to the procedure using the `__old` keyword as follows:

```
__ensures  (__return == __old(*x) + 1)
__modifies (x)
int Foo (int *x) {*x = *x + 1; return *x;}
```

Resources. In addition to the C program expressions, we allow the contracts to refer to "ghost fields" (called *resources*) of objects. Resources are auxiliary fields in data structures meant only for the purpose of specification and manipulated exclusively through contracts. We allow the user to use `__resource(name, expr)` to refer to the value of the ghost field `name` in `expr`. The contract

```
__modifies_resource(name, expr)
```

specifies that the resource `name` is possibly modified at `expr`. Consider the following contract on the procedure `ReleaseNodeA` that releases the `Resource` field of a `NODEA` object:

```
#define __resrA(x) __resource(''NODEA_RES'', x)
#define __modA(x)  __modifies_resource(''NODEA_RES'', x)

#define __releasesA(x)                                \
    __requires (__resrA(x) > 0)                       \
    __ensures  (__resrA(x) == __old(__resrA(x)) - 1)\
    __modA(x)                                         \

__releasesA(NodeA)
void ReleaseNodeA (NODEA NodeA);
```

Type invariants. Figure 4 illustrates type invariants for the `NODEA` and `NODEB` types, using the `__type_invariant` contract. Type invariants specify assertions that

[1] Available at http://research.microsoft.com/projects/havoc/

hold for all objects of a given type. Such invariants typically hold at all control locations except for a handful of procedures where an object is being initialized or being torn down, or may be broken locally inside a basic block (e.g. when an `NODEB` object is added as a child for `NODEA`). The user has the flexibility to specify the control locations where he or she expects the invariants to be temporarily violated.

3.2 Modular checker

In this section, we provide a brief overview of the checker for verifying an annotated procedure. Interested readers can find more details in other works [5]. The main enabling techniques in the checker are:

Accurate memory model for C. `HAVOC` provides a faithful operational semantics for C programs accounting for the low-level operations in systems code. It treats every C pointer expression (including addresses of stack allocated variables, heap locations, and values stored in variables and the heap) uniformly as integers. The heap is modeled as a mutable map or an array `Mem` mapping integers to integers. A structure corresponds to a sequence of pointers and each field corresponds to a compile-time offset within the structure. A pointer dereference `*e` corresponds to a lookup of `Mem` at the address `e` and an update `*x = y` is translated as an update to `Mem` at address `x` with value `y`. Contract expressions are translated in a similar fashion.

Given an annotated C program, the tool translates the annotated source into an annotated `BoogiePL` [12] program, a simple intermediate language with precise operational semantics and support for contracts. The resulting program consists of scalars and maps, and all the complexities of C (pointer arithmetic, & operations, casts etc.) have been compiled away at this stage. Example of the translation can be found in earlier work [6].

Precise verification conditions. `HAVOC` uses the `Boogie` [4] verifier on the generated `BoogiePL` file to construct a logical formula called the *verification condition* (VC). The VC is a formula whose validity implies that the program does not go wrong by failing one of the assertions or the contracts. Moreover, it ensures that the VC generated for a loop-free and call-free program is unsatisfiable *if and only if* the program does not go wrong by failing any assertion or contract present in the code. This is in sharp contrast to most other static analysis tools that lose precision at merge points.

Scalable checking using SMT solvers. The validity of the VC is checked using a state-of-the-art *Satisfiability Modulo Theories* (SMT) solver `Z3` [11]. SMT solvers are extensions of the Boolean Satisfiability (SAT) solvers that handle different *logical theories* such as equality with uninterpreted functions, arithmetic and arrays. These solvers leverage the advances in SAT solving with powerful implementation of theory specific algorithms. These tools can scale to large verification conditions by leveraging conflict-driven learning, smart backtracking and efficient theory reasoning. The modular analysis with efficient SMT solvers provides a scalable and relatively precise checker for realistic procedures up to a few thousand lines large.

3.3 Interprocedural contract inference

HAVOC, like any other procedure-modular checker, requires contracts for called procedures. We have implemented a contract inference algorithm in HAVOC based on the Houdini [16] algorithm in ESC/Java. The algorithm takes as input a partially annotated module along with a finite set of *candidate contracts* for each procedure in the module, and outputs a subset of the candidates that are valid contracts for the module. The candidate contracts are specified by providing an expression inside `__c_requires`, `__c_ensures` and `__c_loop_inv` contracts. For example, the candidate contracts on a procedure `Foo` are shown below:

```
__c_requires (x != NULL)
__c_ensures  (__return > __old(*x))
int Foo (int *x) {*x = *x + 1; return *x;}
```

The Houdini algorithm performs a fixed point algorithm as follows: Initially, the contract for each procedure is the union of the user-provided contracts and the set of candidate contracts. At any iteration, it removes a candidate contract that can be violated during a modular checking of a procedure. The algorithm terminates when the set of candidate contracts does not change.

3.4 Instrumentation

HAVOC also provides different ways for instrumenting the source code with additional contracts (either candidate or normal ones), to relieve the user of manually annotating large modules with similar assertions. The two principle mechanisms of instrumentation are:

- *Access-instrumentation*: The user can direct the tool to add any assertion at every (read, write or both) access to either (i) a global variable, (ii) all objects of a given type, or (iii) fields of objects of a given type.
- *Function-instrumentation*: The user can also direct the tool to add a contract (possibly a candidate contract) to every procedure with a parameter of a given type.

These instrumentations are extremely useful to define properties and thereafter populate candidate contracts of a given kind. For example, to specify that any access to a field `x->f` of an object `x` of given type T is always protected by a lock `x->lock`, we use the *access-instrumentation* feature to add an assertion `x->lock` being held before any access to `x->f`. On the other hand, one can use the *function-instrumentation* feature to populate a class of candidate contracts on all the procedures in a module. For instance, we can add a candidate precondition that the lock `x->ParentA->Resource` is acquired, for any procedure that has a parameter `x` (to be substituted with the formal parameter) of type `NODEB`. Note that in the original implementation in ESC/Java, the Houdini algorithm was used with a fixed set of candidate contracts — namely for checking non-null assertions, index-out-of-bound errors etc. on parameters and return values. The ability to add user-defined candidate contracts is extremely crucial for allowing the user to leverage the contract inference while checking user-defined properties.

4 Challenges and design decisions

In this section, we describe the challenges we faced in applying HAVOC to well-tested codebases of this complexity. We also outline the design decisions that have enabled us to find serious bugs with relatively low false alarms, modest annotation effort and high coverage (particularly on COMP).

4.1 Aliasing

Checking properties that depend on the heap can be difficult because of indirect accesses by pointers; this is because different pointer expressions can evaluate to the same heap location. The problem affects modular checkers as it is not natural to express aliasing constraints as procedure contracts, and may require substantial annotation burden. Finally, the problem is worse for C programs where the addresses of any two fields `&x->f` and `&y->g` can be aliased, due to the lack of type safety. This results in numerous false alarms while checking properties that depend on the heap. We introduce two sources of *justifiable assumptions* that allow us to check the desired properties by separating concerns about type-safety of the program as explicit assumptions.

- **Field safety.** We assume that the addresses of two different *word-type* fields (fields that are not nested structures or unions) can never alias, i.e., `&x->f` and `&y->g` cannot be equal, whenever `f` and `g` are distinct fields. This assumption is mostly maintained with the exception of cases where the program exploits *structural subtyping* whereby two structures with identical layout of types are considered equivalent, even though the field names might differ. The user only needs to specify these exceptions to the tool using additional contracts.
- **Type assumptions.** Many aliasing and non-aliasing constraints can be captured by type invariants similar to the ones shown in Figure 4. These invariants are established after object initialization and are violated at very few places temporarily. The type invariants are currently assumed but not asserted, and help to reduce false positives significantly when dealing with unbounded sets of objects in lists.

Although, both field-safety and the type invariants can be verified in HAVOC [6, 20, 21], they require reasoning with quantifiers and the annotation overhead can be fairly high. Discharging these obligations would improve the confidence in the results of the property checking.

4.2 Modifies clauses

Modifies clauses are used to specify the side-effect of a procedure on the globals and the heap. Specifying a precise set of modified locations for the heap and the resources may require significant annotation burden. On one hand, using coarse-grained modifies information may result in invalidating relevant facts at call sites needed for checking a property; on the other hand, the checker would complain if the specified locations do not contain the locations that are actually modified. Various ownership and encapsulation

methodologies have been proposed [4], but they impose restrictions on the heap manipulation that are often not satisfied by low-level systems code. For soundness, these methodologies impose additional assertions in the program that might require substantial annotation overhead to discharge.

We have found the two following strategies to achieve a low annotation overhead without sacrificing significant coverage.

Property state modifies: To keep the annotation burden low for checking, we decided to make the modifies clauses for the heap unchecked, i.e., they are assumed at the call sites, but not checked as postconditions. However, for the resources in the property, we require the user to specify sound modifies clauses. Although this introduces unsoundness in our checking and may suppress real bugs, we found it to be pragmatic tradeoff based on the following observation: most of the pointer fields in the program that point to other objects in the heap and define the *shape* of data structures are immutable with very few exceptions. For instance, the `ParentA` in a `NODEB` object is set after initialization and remains immutable afterwards. A quick grep revealed that the `ParentA` field in a `NODEB` object is read at least in 1100 places in the source, however it is written to at only 8 places, mostly in the creation path. For fields like `ReferenceCount` in `NODEA` objects that form part of a property, we maintain a resource to track the value of this field, and thereby support sound modifies clauses.

OUT parameter modifies: Making the modifies clause *free* for fields in the heap almost allowed us to avoid specifying modifies clauses for the fields in the heap. However, we found the need for specifying modifies clauses for *out* parameters of a procedure to avoid the following situation that is quite common in systems code:

```
void Bar(.., PSCB *LocalScb);

void Foo(...){
        PSCB LocalScb = NULL;
        ....
        Bar(..., &LocalScb);
        ...
        if (LocalScb){...}
        ...
}
```

If we do not provide a modifies clause for `Bar` to indicate that the heap has changed at the location `&LocalScb`, the checker would assume the code inside the **then**-branch of "`if(LocalScb)`" is unreachable, and therefore be unsound. To avoid this, we used the contract inference to infer modifies clauses for the parameters that are used as out parameters.

4.3 Interactive contract inference

The typical use of the contract inference engine was to infer a set of simple contracts that would hold for a large number of procedures, possibly with a few exceptions. The inference relieves the user by finding the exception set without having to manually inspect the complex call graph. For example, for checking data-race freedom, we inferred the set of procedures where the lock `Resource` in a `NODEA` object is held. This can be achieved by creating candidate contracts about this lock being held on all procedures that have either a `NODEA` or a `NODEB` as a parameter or return value.

```
void CreateChild(PNODEA NodeA, ATTRIBUTE attr,...){
  PNODEB NodeB;
  AcquireNodeAExcl(NodeA);
  CreateNodeB(NodeA, &NodeB,..);
  Initialize1(NodeB, attr,...);
  ...
}

__ensures((*PNodeB)->ParentA == NodeA)
void  CreateNodeB(PNODEA NodeA, PNODEB *PNodeB,..);

void Initialize1(PNODEB NodeB, ..){

  <modify ParentA, State fields in NodeB >
  Initialize2(NodeB, ...);
}

void Initialize2(PNODEB NodeB,..){
  <modify ParentA, State fields in NodeB>
  Initialize3(NodeB, ...);
}
```

Fig. 5. Procedure calls chains

However, the precision of the inference crucially depends on the existing contracts. These contracts could have been manually specified or inferred previously. An attempt to infer contracts without being cognizant of other constraints on the module can lead to significant loss of precision. Consider the Figure 5, where the procedure `CreateChild` creates a child of `NodeA` in `CreateNodeB` and then initializes different parts of the child object and other data structures through several layers of deeply nested calls. Suppose we are interested in inferring the procedures where the `Resource` in an `NODEA` object is held, to check for data-race freedom. Unless the contract on `CreateNodeB` is already specified, the inference engine fails to discover that `NodeB->ParentA->Resource` is held at entry to all the `InitializeX` procedures. The contract on `CreateNodeB` is more difficult to infer since it involves two objects `PNodeB` and `NodeA`.

Therefore, the process of adding manual contracts and applying inference was coupled with the feedback from each step driving the other.

4.4 Exceptions

COMP (and several other modules in Windows) uses *Structured Exception Handling* (SEH) to deal with flow of control due to software and hardware exceptions. In SEH, the program can use either `__try`/`__except` blocks to implement an exception handler, or `__try`/`__finally` blocks to deal with cleanup along both normal and exceptional paths.

```
__try{
   //guarded code
} __except (expr) {
   //exception handler
   //code
}
```

```
__try{
   //guarded code
} __finally{
   //termination code
}
```

To model exceptions, we introduced a resource variable `__thrown` to denote whether a procedure call raises an exception. The variable is reset to `FALSE` at entry to any procedure, is set to `TRUE` whenever a kernel procedure that could raise an exception (e.g.

KeRaiseStatus or ExAllocatePoolWithTag) returns with an exception, and is reset to FALSE once the exception is caught by an exception handler in __except. We introduced a new contract macro:

```
#define __may_throw(WHEN)  __ensures(!WHEN ==> !__thrown)
```

A procedure with a __may_throw(WHEN) contract denotes that the procedure *does not* raise an exception if the condition WHEN does not hold at exit from the procedure. This allows specifying __may_throw(TRUE) on one extreme to indicate that any call to the procedure may throw an exception, and __may_throw(FALSE) on the other extreme to indicate that the procedure *never* raises an exception. Every procedure in the module also has a default modifies clause saying that __thrown can be modified by the procedure.

The presence of exceptions increases the number of paths through a procedure, since any called procedure can potentially throw an exception and jump to the exit. Our initial attempt at ignoring the exceptional paths revealed very few bugs, signifying the well-tested nature and the maturity of the codebase.

To circumvent the problem, we used the inference engine to infer the set of procedures in this module that do not raise an exception. We first annotated the kernel procedures like KeRaiseStatus with __may_throw(WHEN) to denote the constrains on its inputs WHEN under which the procedure may throw an exception. Next, we added a candidate contract __may_throw(FALSE) to each procedur. The interprocedural inference algorithm removes __may_throw(FALSE) from procedures that may potentially raise an exception. The set of procedures on which __may_throw(FALSE) is inferred denotes the procedures that never throw an exception. To improve the precision of inference, we had to manually add contracts for internal procedures that could raise an exception only under certain conditions.

5 Property checking on COMP

5.1 COMP

In this section, we briefly describe the core driver COMP from the Windows®operating system, and the synchronization protocol that was checked. For the sake of security, we keep the component and the names of the procedures anonymous. The component has around 300 thousand lines of code, excluding the sources for the kernel procedures. There are more than 1500 procedures present in the module. The code for the component has evolved over almost two decades, and each new generation inherits a lot of the code from the previous versions. Some of the procedures in the module have up to 4,000 lines of code, signifying the complexity and the legacy nature of the code base. COMP also heavily employs the Microsoft *Structured Exception Handling* (SEH) mechanism for C/C++ to deal with flow of control due to exceptions (discussed more in Section 4.4).

We first provide a brief description of the synchronization protocol governing the management of the main heap-allocated structures in COMP. We will focus on four main type of objects: NODE that is the root type which can contain multiple instances of NODEA, NODEB and NODEC types.

Each NODE has an ERESOURCE field NodeResource and a mutex NodeMutex for synchronization. The ERESOURCE structure implements a reader-writer lock in Windows that can be recursively acquired. The NodeResource acts as a global lock for access to any NODEA, NODEB and NODEC objects within a given NODE (i.e. it is sufficient to acquire this lock to access any field in the NODEA, NODEB and NODEC objects).

Each NODEA object has a list of NODEB children (as described in Section 2) and a list of NODEC children. Each NODEA has a ERESOURCE field Resource that protects most of its fields and the fields of its children NODEB and NODEC objects; each NODEA also has a mutex NodeAMutex that protects a set of other fields in each NODEA and its NODEB and NODEC children.

Each NODEA also has an integer field ReferenceCount that signifies the number of threads that have a handle on a particular NODEA object — a positive value of ReferenceCount on an NODEA object indicates that some thread has a handle on the object and therefore can't be freed.

There is a global list ExclusiveNodeAList of all the NODEA objects for which the Resource has been acquired. A call to the procedure ReleaseNodeAResources releases the Resource field of any NODEA on the ExclusiveNodeAList.

5.2 Properties

COMP has a synchronization protocol governing the creation, usage and reclamation of the objects in a multi-threaded setting. The synchronization is implemented by a combination of reference counting, locks and other counters in these objects, and is specific to this module. The integrity of the protocol depends on several properties whose violations can lead to serious bugs:

1. Ref-count usage. We checked that for every execution path, the increments and decrements of the ReferenceCount field of a NODEA object are balanced. Decrementing the count without first incrementing could lead to freeing objects in use and a net increment in this field would correspond to a resource leak, as the NODEA object will not be reclaimed.

2. Lock usage. We check for the violation of the locking protocol for the various locks in NODE and NODEA objects. For a mutex field, we check that the lock is acquired and released in alternation; for a reader-writer lock which can be acquired recursively, we check that each release is preceded by an acquire.

3. Data race freedom. This is roughly the property that we described in Section 2, except that we monitor reads and writes for the other fields in these objects too. Since the NodeResource in a NODE object acts a global lock, we need the Resource field in a NODEA object be held only when the global NodeResource lock is not held.

4. Teardown race freedom. We check for races between one thread freeing a NODEA object, and another thread accessing the same object. Any thread freeing a NODEA object must hold that NODEA's Resource exclusive, hold the parent NODE's NodeMutex, and ensure that NODEA's ReferenceCount is zero. Conversely, any thread accessing a NODEA must *either* hold the NODEA's Resource shared or exclusive, hold the parent NODE's NodeMutex, or have incremented the ReferenceCount field. These rules ensure mutual exclusion between threads freeing and accessing NODEA

Annotations	LOC
Property	250
Manual	600
Inferred	3000
Total	3850

Property	# of bugs
Ref-count	14
Lock usage	12
Data races	13
Teardown	6
Total	45

Fig. 6. Annotation overhead and bugs.

objects, and any rule violation could lead to a teardown race. This is a domain-specific property which requires the user to define the property.

5.3 Results

In this section, we describe our experience with applying HAVOC on COMP. Figure 6 summarizes the annotation effort and the distribution of the 45 bugs found for the four properties listed above. The "Property" annotations are specifications written to describe the property and also to specify the behavior of kernel procedures. The "Manual" annotations correspond to procedure contracts, loop invariants and type invariants for this module. Finally, the "Inferred" annotations are a set of contracts that are automatically generated by the contract inference described in Section 3.3.

Currently, our checker runs on the annotated code for COMP, and generates 125 warnings over the approximately 1500 procedures in 93 minutes — this corresponds to roughly 3.7 seconds spent analyzing each procedure on average. Most of the runtime (roughly 70%) is spent in a non-optimized implementation for converting C programs into BoogiePL programs, which can be significantly improved. Further, each source file (roughly 60 of them in COMP) in the module can be analyzed separately, and hence the process can be easily parallelized to reduce the runtime.

Out of the 125 warnings, roughly one third of the warnings correspond to confirmed violations of the four properties listed above. This is a fairly low false positive rate, given that we have not invested in various domain-specific filters to suppress the unlikely bugs.

In the following sections, we discuss details of a few bugs, the breakup of the manual annotations and the inferred annotations, and the assumptions that might lead to missed bugs.

5.4 Bugs found

In this section, we describe two representative bugs from the set of 45 violations to the different properties. An interesting nature of most of the bugs is that they appear along exceptional paths — paths where some procedure raises an exception. This suggests the maturity and well-tested nature of the code as well as the fact that HAVOC can find these subtle corner cases. Besides, some of these synchronization bugs are hard to reproduce in a dynamic setting; the developers of the codebase suspected a leak in the ReferenceCount field but had been unable to reproduce it.

```
...
__try{
  ...
  NodeA = CreateNodeA(Context, ..);

  if (!AcquireExclNodeA(Context, NodeA, NULL, ACQUIRE_DONT_WAIT )) {

      NodeA->ReferenceCount += 1;
      ...
      AcquireExclNodeA(Context, NodeA, NULL, 0 );
      ...
      NodeA->ReferenceCount -= 1;
  }
  ...
} __finally {
  ...
}
...
```

Fig. 7. Reference count leak.

```
...
if (!AcquireExclNodeA(Context, NodeA, NULL, ACQUIRE_DONT_WAIT)) {
    ...
    AcquireExclNodeA(Context, NodeA, NULL, 0);
    ...
}

SetFlag(NodeA->NodeAState, NODEA_STATE_REPAIRED);
...
PerformSomeTask(Context, ...);
...
if (FlagOn( ChangeContext.Flags, ... )) {
    UpdateNodeAAndNodeB(Context, NodeA, ChangeContext.Flags);
}
...
```

Fig. 8. Data race on NODEA object.

Reference count leak. Figure 7 illustrates an example of a bug that leads to a violation of the Ref-count usage property. In the example, an object NodeA of type NODEA is created in CreateNodeA and then an attempt is made to acquire the Resource in NodeA using the procedure AcquireExclNodeA. This procedure has the behavior that it can return immediately or perform a blocking wait on the Resource depending on whether the flag ACQUIRE_DONT_WAIT is specified or not. Hence, if the first non-blocking acquire fails in the if statement, then it tries a blocking acquire. Before doing that, it increments the ReferenceCount field to indicate a handle on this NODEA object; the field is decremented once the Resource is acquired. However, if AcquireExclNodeA throws an exception, then the __finally block does not decrement the ReferenceCount field, and hence this NODEA object will always have a spurious handle and will never be reclaimed.

Data-race. Figure 8 illustrates an example of data-race on the fields of NODEA object. The procedure first acquires the Resource lock of an object NodeA in the first if block. The fields of NodeA are modified in the SetFlag macro and in the UpdateNodeAAndNodeB procedure. The access in SetFlag is protected by the Resource lock. However, the procedure PerformSomeTask calls the procedure ReleaseNodeAResources transitively with a deeply nested call chain, which might release the Resource lock in any NODEA object. This means that the Resource lock

is not held at entry to `UpdateNodeAAndNodeB`, although the procedure expects this lock to be held at entry to modify the fields of `NodeA`.

5.5 Manual contracts

We classify the main source of manual contracts in this section. In addition to the aliasing constraints and type invariants described in Section 2, we also annotated a variety of interesting conditional specifications and loop invariants.

Conditional specifications. Consider procedure `AcquireExclNodeA` that was present in the two bugs described in Section 5.4 and its contract:

```
__acquire_nodeA_excl(NodeA, !__thrown && __return != FALSE)
__ensures(!FlagOn(Flags, ACQUIRE_DONT_WAIT) && !__thrown
                         ==> __return != FALSE)
BOOLEAN AcquireExclNodeA (PCONTEXT Context,
                          PNODEA NodeA, PNODEB NodeB, ULONG Flags);
```

Recall (from Section 4.4) that `__thrown` indicates whether a procedure has a normal return or an exceptional return. The first annotation (an annotation macro composed of `__requires`, `__ensures` and `__modifies`) describes the condition under which the `Resource` field of `NodeA` parameter is acquired. The second annotation specifies that if `ACQUIRE_DONT_WAIT` flag is not set, and the procedure does not throw an exception, then the return value is never `FALSE`.

Loop invariants. We also specified loop invariants when the property being checked depends on state modified inside a loop. The procedure `ClearChild` in Figure 3 provides an example of such a loop invariant. But a more common form of loop invariant arises due to the following code pattern:

```
BOOLEAN TryAcquireNodeA(PNODEA NodeA,..)
{
    BOOLEAN AcquiredFlag = FALSE;
    ...
    __try{
       ...
       __loop_inv(AcquiredFlag == FALSE)
       while (true) {
            CallMightRaise1();
            if (..){
               AcquireNodeAExcl(NodeA);
               AcquiredFlag = TRUE;
               CallMightRaise2();
               return TRUE;
            }
       }
    } __finally {
       ...
       if (AcquiredFlag)
           ReleaseNodeA(NodeA);
       ...
       return FALSE;
    }
}
```

The callers of `TryAcquireNodeA` expect that the procedure acquires the resource of `NodeA` at normal exit. However, in the absence of the loop invariant, the checker would report a false warning where the `ReleaseNodeA` tries to release a resource

Contracts type	# of inferred annot
May throw	914
`NodeResource` held	107
`NodeMutex` not held	674
`NODEAResource` held	360
`NODEAResource` release all	210
OUT parameter modified	271
Parameter flag set	331
Total	2867

Fig. 9. Distribution of inferred contracts.

without first acquiring it. This happens because in the absence of the loop invariant, the checker will report a path where the value of `AcquiredFlag` is `TRUE` at the loop head, the procedure `CallMightRaise1` throws an exception and control reaches the `_finally` block.

5.6 Inferred contracts

`HAVOC`'s automatic inference capability generated a majority of the simple contracts (around 3000 of them) and was crucial to the automation of the tool for such a complex codebase (i.e. only 600 manually written contracts on around 1500 functions analyzed by the tool).

Figure 9 summarizes the main classes of contracts that were generated using the automated inference mechanism. In addition to the inference about `__may_throw` contracts and modifies clauses for the out parameters of a procedure, we employed the inference engine to infer a certain type-state property on some objects of type `NODEA` or `NODEB` on the procedures in the module.

1. **May throw:** as described in Section 4.4, this denotes the set of procedures that do not raise an exception.
2. `NodeResource` **held:** infers a set of procedures where the lock `NodeResource` on the global `NODE` object is held at entry to ensure data-race freedom.
3. `NodeMutex` **not held:** infers a set of procedures where the `NodeMutex` field of the global `NODE` is not held at entry. Since most procedures acquire and release this lock locally inside a procedure, this contract is useful for proving that locks are not acquired twice.
4. `NODEAResource` **held:** infers that the `Resource` field for an `NODEA` parameter or the `Resource` field for the parent of an `NODEB` or `NODEC` object is held at entry to a set of procedures. This along with `NodeResource` ensures absence of data-races.
5. `NODEAResource` **release all:** infers the set of procedures that could release the `Resource` of any `NODEA` object by a transitive call to `ReleaseNodeAResources`.
6. **OUT parameter modified:** adds a `_modifies(x)` contract for an out parameter `x` that is modified inside a procedure, as described in Section 4.2.

7. **Parameter flag set:** infers a set of procedures where a certain field of a parameter is set to `TRUE` on entry to the procedures. The parameter captures the state of computations that span multiple procedures and is threaded through the nested procedure calls. The parameter `Context` in Figures 7 and Figure 8 is an example of such a parameter.

5.7 Assumptions

HAVOC provides a set of options that allows the user to introduce a class of *explicit* assumptions into the verification, which can be enumerated and discharged later with more contracts or a separate analysis. This allows the user of the tool to control the degree of unsoundness in the verification, and to recover from them using more contracts. This is in contrast to most other static analysis tools that bake these assumptions into the analysis and there is no way to recover from them. There are three main sources of such assumptions in our current analysis: (1) field safety, (2) type invariant assumptions and (3) free modifies for the heap fields. The first two sources were discussed in Section 4.1 and the third in Section 4.2.

Of the three options, we believe that both field safety and the type invariants hold for the module with very few exceptions and separate the proof of the high-level properties from the proofs of type-safety and type/shape invariants. Eliminating the free modifies clauses for the heap fields are the assumptions that we would like to eliminate to increase the confidence in the checking.

5.8 False warnings

As mentioned earlier, the tool generates a total of 125 warnings, and roughly one third of the warnings correspond to confirmed violations of the four properties listed above. Unlike typical static analyzers, the remaining warnings are not violation of the properties being checked. Instead, most of these warnings are violations of intermediate procedure contracts which were used to discharge the properties of interest.

Of course, the soundness of a modular proof can be compromised by the presence of even a single warning. However, for large code bases, it is very difficult to verify *every* contract. To obtain a balance, we require that the remaining warnings are not violations of automatically generated assertions (for the property), but rather violation of user-specified contracts. The rationale being that user provided contracts are a result of understanding the code, and have a good chance of being true (although they may not be inductive). However, proving these contracts require adding contracts on other fields of these structures, or devising a new template for contracts (e.g. checking which fields of an object are non-null).

6 Security audit

In this section, we briefly describe preliminary experience of applying HAVOC for checking security vulnerabilities in some of the modules in Microsoft Windows. In spite

of extensive testing and the application of customized static analysis on these components, these components still have bugs that make them vulnerable to malicious attacks. Careful code audit is essential to safeguard the systems against such attacks, but manual inspection is expensive and error-prone.

HAVOC has been deployed to check several properties whose violation can often lead to exploitable attacks:

- *ProbeBeforeUse*: any pointer that can be passed by a user application (*user* pointers) to the kernel APIs must undergo a call to a procedure *Probe* before it is dereferenced.
- *UserDerefInTry*: any dereference of a user pointer must happen inside a `__try` block,
- *ProbeInTry*: any call to *Probe* should happen inside a `__try` block to correctly deal with cases when a user passes illegal pointers,
- *Alloc0*: ensure that the non-null buffer returned by calling *malloc* with a size of zero is handled safely. Although it is legal to call allocation procedures with a size of zero, such allocations often lead to buffer overruns without proper safeguard [23].

Although the properties are simple, it is non-trivial to ensure the absence of these bugs primarily due to deep call chains and the presence of deeply nested pointers. For example, to check either the *ProbeBeforeUse* or *UserDerefInTry*, one needs to know whether any pointer that is dereferenced in the program can alias with one of the pointers that are reachable from the globals or parameters to the entry functions of the module. There can be several thousand dereferences in a large module and validating each of them (especially those in deeply nested procedures) can be challenging. On the other hand, the *Alloc0* property requires arithmetic reasoning as the allocation size could be 0 because of an overflow.

The properties were specified quite easily by adding suitable contracts to *Probe* and *malloc* procedures. We have analyzed several modules (with more than a million lines across all of them) for various subset of these properties. We have created various inference for interprocedural reasoning including (a) propagation of information about the pointers that have undergone a call to *Probe*, (b) the procedures that are always called from within a `__try` block, etc. Details of the inference and results are outside the scope of this article, since this is a work in progress. In addition, the user had to provide some annotations (in the order of a few hundred currently) manually. The effort has lead to the discovery of four vulnerabilities related to *ProbeBeforeUse* and around ten vulnerabilities related to *Alloc0*, all of which have been patched. The tool allows the auditor to only inspect around 2-3% (for modules with around 3000 procedures) of all procedures for warnings for the *ProbeBeforeUse* properties and around 10% of the allocation sites for the *Alloc0* sites. We are working to further reduce these false alarms with better inference, loop invariants etc. However, the ability for the user to construct these property checkers and guide the inference to use the domain knowledge has provided value in focusing the time of an auditor on the more problematic procedures.

7 Related Work

There is a rich literature on static analysis tools for finding various defects in software programs. We discuss some of these tools in this section, to perform a qualitative analysis of the strengths and weaknesses of using these tools for our case study.

Contract-based checkers. HAVOC is closely based on the principles of ESC/Java [17] tool for Java programs and Spec# [4] tool for C# programs. The main difference lies in our intent to analyze systems program written in C, that requires support for low-level operations in both the source and the contract language. Secondly, although ESC/Java was applied to real life Java programs to demonstrate the usefulness of contract inference [16, 17], the tool did not allow the user to create customizable inference for particular contracts. These tools have not been applied to real progams of the scale considered in this paper to find bugs in a cost-effective manner with low annotation overhead.

SAL in an annotation language for documenting buffer related properties for C programs and espX is a checker for the language [18]. This is one of the few examples of annotation based checker for a specific property. The language is not extensible, and does not allow specifying new user-defined properties.

Dedicated property checkers. A majority of the numerous static analysis tools developed for systems software in the last decade fall in this category — we highlight only a representative sample for the different properties that scale to several thousand lines of code. Examples of data-race checkers include Relay [24], LOCKSMITH [22], RacerX [13]. CALYSTO [2] finds null dereference bugs in C programs by using SAT solvers. The ASTREÉ analyzer [8] uses abstract interpretation [7] to prove the absence of certain runtime errors such as buffer overruns, integer overflows in embedded safety-critical software. Most of these tools do not require user annotations, use novel algorithms based on data-flow analysis, often with the intent of finding bugs at the cost of unsound assumptions.

Extensible property checkers. Tools such as SLAM [3], BLAST [19] and ESP [10] are examples of software model checkers that check a property by exhaustively analyzing models of C programs. Their property languages allow specifying simple state-machines over the typestate of objects, and can express simple lock usage properties. These tools are most suited for checking properties on global variables, and lose precision and soundness when dealing with low-level operations and relationships between objects in the heap. Our case study shows the need for both in checking the synchronization protocol.

Meta-level compilation [14] provides compiler extensions to encode patterns of violations for system-specific properties in a state-machine language *metal*, which are checked at compile time. The technique finds serious errors in systems code, but does not attempt to maintain soundness or guarantees about the absence of such bugs. These tools are suitable for describing bug patterns in a code, but once again are poorly suited for describing detailed properties of the heap (for example the absence of teardown race).

Saturn [1] uses a logic programming framework to specify static analysis. Saturn also uses a concrete operational semantics similar to HAVOC. While HAVOC's meta-theory is fixed and based on contracts, the meta-theory of Saturn may be extended

by analyses expressed in a logic programming language. The ability to add inference rules adds flexibility in analysis design but comes at two different costs. First, extending `Saturn` requires an expert analysis designer whereas extending `HAVOC` could be done by a programmer simply by the use of contracts. Second, the meta-theory behind the analyses is usually not proved correct and could therefore introduce unexpected unsoundness into the system.

8 Conclusions

In this work, we have demonstrated the feasibility of applying contract-based checkers for scalable user-defined property checking, and the challenges involved in scaling such an approach to large code bases with modest annotation overhead, low false alarms, without sacrificing a lot of coverage. Our work points out several immediate directions of future work that would improve the usability of modular checkers such as `HAVOC` in the hand of a user: better inference of conditional contracts can relieve a lot of annotation burden, inference of modifies clauses will allow us to remove unsoundness issues related to the unchecked modifies clauses, and finally, we need easy-to-use annotations for specifying invariants at the level of types.

References

1. A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins. An overview of the Saturn project. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE '07)*, pages 43–48, 2007.
2. D. Babic and A. J. Hu. Structural abstraction of software verification conditions. In *Computer Aided Verification (CAV '07)*, LNCS 4590, pages 366–378, 2007.
3. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI '01)*, pages 203–213, 2001.
4. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS '05)*, LNCS 3362, pages 49–69, 2005.
5. S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamarić. A reachability predicate for analyzing low-level software. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '07)*, LNCS 4424, pages 19–33, 2007.
6. J. Condit, B. Hackett, S. K. Lahiri, and S. Qadeer. Unifying type checking and property checking for low-level code. In *Principles of Programming Languages (POPL '09)*, pages 302–314, 2009.
7. P. Cousot and R. Cousot. Abstract interpretation : A Unified Lattice Model for the Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Principles of Programming Languages (POPL '77)*, pages 238–252, 1977.
8. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTREÉ Analyzer. In *European Symposium on Programming (ESOP '05)*, LNCS 3444, pages 21–30, 2005.
9. M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. Vcc: Contract-based modular verification of concurrent c. In *International Conference on Software Engineering, (ICSE '09), Companion Volume*, pages 429–430, 2009.

10. M. Das, S. Lerner, and M. Seigle. ESP: Path-Sensitive Program Verification in Polynomial Time. In *Programming Language Design and Implementation (PLDI '02)*, pages 57–68, 2002.
11. L. de Moura and N. Bjorner. Efficient Incremental E-matching for SMT Solvers. In *Conference on Automated Deduction (CADE '07)*, LNCS 4603, pages 183–198, 2007.
12. R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.
13. D. R. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Symposium on Operating Systems Principles (SOSP '03)*, pages 237–252, 2003.
14. D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Operating Systems Design And Implementation (OSDI '00)*, pages 1–16, 2000.
15. C. Flanagan and S. N. Freund. Type-based race detection for java. In *PLDI*, pages 219–232, 2000.
16. C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *International Symposium of Formal Methods Europe (FME '01)*, pages 500–517, 2001.
17. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Language Design and Implementation (PLDI'02)*, pages 234–245, 2002.
18. B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *International Conference on Software Engineering (ICSE '06)*, pages 232–241, 2006.
19. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Principles of Programming Languages (POPL '02)*, pages 58–70, 2002.
20. S. K. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In *Principles of Programming Languages (POPL '08)*, pages 171–182, 2008.
21. S. K. Lahiri, S. Qadeer, J. P. Galeotti, J. W. Voung, and T. Wies. Intra-module inference. In *Computer Aided Verification (CAV '09)*, LNCS 5643, pages 493–508, 2009.
22. P. Pratikakis, J. S. Foster, and M. W. Hicks. LOCKSMITH: context-sensitive correlation analysis for race detection. In *Programming Language Design and Implementation (PLDI '06)*, pages 320–331, 2006.
23. J. Vanegue. Zero-sized heap allocations vulnerability analysis.
24. J. W. Voung, R. Jhala, and S. Lerner. RELAY: static race detection on millions of lines of code. In *Foundations of Software Engineering (FSE '07)*, pages 205–214, 2007.