Saturn: A SAT-Based Tool for Bug Detection*

Yichen Xie and Alex Aiken

Computer Science Department
Stanford University
{yxie, aiken}cs.stanford.edu

1 Introduction

Saturn is a boolean satisfiability (SAT) based framework for static bug detection. It targets software written in C and is designed to support a wide range of property checkers.

The goal of the Saturn project is to realize SAT's potential for precise checking on very large software systems. Intraprocedurally, Saturn uses a bit-level representation to faithfully model common program constructs. Interprocedurally, it employs a summary-based modular analysis to infer and simulate function behavior. In practice, this design provides great precision where needed, while maintaining observed linear scaling behavior to arbitrarily large software code bases. We have demonstrated the effectiveness of our approach by building a lock analyzer for Linux, which found hundreds of previously unknown errors with a lower false positive rate than previous efforts [16].

The rest of the paper is organized as follows. Section 2 gives an overview of the Saturn analysis framework. Section 3 describes the modeling of common program constructs in Saturn. Section 4 describes the lock checker for Linux. We discuss related work in Section 5 and our conclusions in Section 6.

2 Overview

The Saturn framework consists of four components: 1) a low-level Saturn *Intermediate Language* (SIL) that models common program constructs such as integers, pointers, records, and conditional branches, 2) a CIL-based [14] *frontend* that parses C code and transforms it into SIL, 3) a SAT-based *transformer* that translates SIL statements and expressions into boolean formulas, and 4) property *checkers* that infer and check program behavior with respect to a specific property.

A Saturn analysis proceeds as follows:

– First, we use the frontend to parse C source files and transform them into SIL. The resulting abstract syntax trees are stored in a database indexed by file and function names.

* Supported by NSF grant CCF-0430378.

- Second, we construct the static call graph of the program. The call graph is then sorted in topological order (callee first). Strongly connected components (SCC) in the call graph are collapsed into supernodes that represent the collection of functions in the SCC.
- Third, the property checker retrieves and analyzes each function in the codebase in topological order. (Property checkers determine how call graph cycles are handled; currently our analyses simply break such cycles arbitrarily.) It infers and checks function behavior with respect to the current property by issuing SAT-queries constructed from the boolean constraints generated by the transformer. The inferred behavior is then summarized in a concise representation and stored in the summary database, to be used later in the analysis of the function's callers.
- Finally, violations of the property discovered in the previous step are compiled into bug reports. The summary database is also exported as documentation of the inferred behavior of each function, which is immensely helpful during bug confirmation.

3 The Saturn Intermediate Language

In this section, we briefly highlight the program constructs supported by the Saturn Intermediate Language (SIL). The formal definition of SIL and the details of its translation to the boolean representation are described in [16].

Integers. Saturn models n-bit signed and unsigned integers by using bit-vector representations. Signed integers are expressed using the 2's complement representation and common operations such as addition, subtraction, comparison, negation, and bitwise operations are modeled faithfully by constructing boolean formulas that carry out the computation (e.g., a ripple carry adder). More complex operations such as division and remainder are modeled approximately.

Pointers. Saturn supports pointers in C-like languages with two operations: *load* and *store*. We use a novel representation called *guarded location sets* (GLS), defined as a set of pairs (g, l) where g is a boolean guard and l is an abstract location. GLS track the set of locations that a pointer can point to and the condition under which the points-to relationship holds. This approach provides a precise and easily accessible representation for the checker to obtain information about the shape and content of a program's heap.

Records. Records (i.e., `struct`s in C) in Saturn are modeled as a collection of component objects. Supported operations include field selection (e.g. `x.state`), dereference (e.g. `p->data.value`) and taking an address through pointers (e.g. `&curr->next`).

Control flow. Saturn supports programs with reducible control flow graphs.[1] Loops are modeled by unrolling a predetermined number of times and discarding the backedges. The rationale of our approach is based on the observation that

[1] Non-reducible control flow are rare (0.05% in the Linux kernel), and can be transformed into reducible ones by node-splitting [1].

many errors have simple counterexamples, and therefore should surface within the first few iterations of the loop; this approach is essentially an instance of the *small scope hypothesis* [12]. Compared to abstraction based techniques, unrolling trades off soundness for precision in modeling the initial iterations of the loop. Our experiments have shown that for the properties we have checked, unrolling contributes to the low false positive rate while missing few errors compared to sound tools.

Function calls. We adopt a modular approach to modeling function calls. Saturn analyzes one function at a time, inferring and summarizing function behavior using SAT queries and expressing the behavior in a concise representation. Each call site of the function is then replaced by instrumentation that simulates the function's behavior based on the summary. This approach exploits the natural abstraction boundary at function calls and allows Saturn to scale to arbitrarily large code bases.[2] The summary definition is checker specific; we give a concrete example in the following section.

4 Case Study: A Lock Checker for Linux

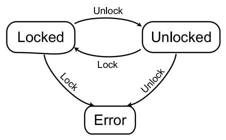

Fig. 1. FSM for locks

To experimentally validate our approach, we have developed a checker using the Saturn framework that infers and checks locking behavior in Linux. Locks are a classic example of a temporal safety property and have been the focus of several previous studies [8, 7, 3]. Locking behavior for a single lock in a single thread is described by the finite state machine (FSM) shown in Figure 1. Re-locking an already locked object may result in a system hang, and releasing an unlocked object also leads to unspecified behavior. Our checker targets such violations.

We model locks using SIL constructs. We use integer constants to represent the three states `locked`, `unlocked`, and `error`, and we attach a special `state` field to each lock object to keep track of its current state. State transitions on a lock object are modeled using conditional assignments. We show an example of this instrumentation below:

```
void lock_wrapper(lock_t *l) {          void lock_wrapper(lock_t *l) {
  lock(l);                         ⇒      if (l−>state == UNLOCKED)
}                                            l−>state = LOCKED;
                                           else
                                             l−>state = ERROR;
                                        }
```

For the `lock` operation above, we first ensure the current state is `unlocked`. If so, the new state is `locked`; otherwise, the new state is `error`. Every call to `lock` is replaced by this instrumentation. The instrumentation for `unlock` is similar.

Using this instrumentation for locks, we infer the locking behavior of a function `f` by issuing SAT queries for each possible pair of start and finish states

[2] The lock checker we describe in Section 4 averages 67LOC/s over nearly 5M lines of Linux.

of each lock `f` uses. If the query is satisfiable, then there is a possible transition between that pair of states across the function. In the example above, the satisfiable pairs are `unlocked` $\rightarrow$ `locked` and `locked` $\rightarrow$ `error`. We record the set of possible transitions in the summary database and use it later when we analyze the callers of the function. To check a function's locking behavior, we check that there is at least one legal transition (i.e. one that does not end in the `ERROR` state) through the function.

We have run the lock checker over Linux, which contains roughly 5 million lines of code. Our analysis finished in about 20 hours and issued 300 warnings, 179 of which are believed to be real errors by manual inspection.

5 Related Work

Jackson and Vaziri were apparently the first to consider finding bugs via reducing program source to boolean formulas [12]. Subsequently there has been significant work on a similar approach called *bounded model checking* [13, 4, 11]. While there are algorithmic differences between SATURN and these other systems, the primary conceptual difference is our emphasis on scalability (e.g., function summaries) and focus on fully automated checking of properties without separate programmer-written specifications.

Static analysis tools commonly rely on abstraction techniques to simplify the analysis of program properties. SLAM [3] and BLAST [10, 9] use predicate abstraction techniques to transform C code into boolean programs. ESP [5] and MC [8, 6] use the finite state machine (FSM) abstraction and employ interprocedural dataflow analyses to check FSM properties. CQual [7, 2] is a type-based checking tool that uses flow-sensitive type qualifiers to check similar properties. In contrast, a SAT-based approach naturally adapts to a variety of abstractions, and therefore should be more flexible in checking a wide range of properties with precision.

Several other systems have investigated encoding C pointers using boolean formulas. CBMC [13] uses uninterpreted functions. SpC [15] uses static points-to sets derived from an alias analysis. There, the problem is much simplified since the points-to relationship is concretized at runtime and integer tags (instead of boolean formulas) can be used to guard the points-to relationships. F-Soft [11] models pointers by introducing extra equivalence constraints for all objects reachable from a pointer, which is inefficient in the presence of frequent pointer assignments.

6 Conclusion

We have presented SATURN, a scalable and precise error detection framework based on boolean satisfiability. Our system has a novel combination of features: it models all values, including those in the heap, path sensitively down to the bit level, it computes function summaries automatically, and it scales to millions of lines of code. We demonstrate the utility of the tool with a lock checker for Linux, finding in the process 179 unique locking errors in the Linux kernel.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.
2. A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and inferring local non-aliasing. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 129–140, June 2003.
3. T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of SPIN 2001 Workshop on Model Checking of Software*, pages 103–122, May 2001. LNCS 2057.
4. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.
5. M. Das, S. Lerner, and M. Seigle. Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.
6. D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, Sept. 2000.
7. J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 1–12, June 2002.
8. S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.
9. T. A. Henzinger, R. Jhala, and R. Majumdar. Lazy abstraction. In *Proceedings of the 29th Annual Symposium on Principles of Programming Languages (POPL)*, January 2002.
10. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast. In *Proceedings of the SPIN 2003 Workshop on Model Checking Software*, pages 235–239, May 2003. LNCS 2648.
11. F. Ivancic, Z. Yang, M. Ganai, A. Gupta, and P. Ashar. Efficient SAT-based bounded model checking for software verification. In *Proceedings of 1st International Symposium on Leveraging Applications of Formal Methods (ISoLA)*, 2004.
12. D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2000.
13. D. Kroening, E. Clarke, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of the 40th Design Automation Conference*, 2003.
14. G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the 11th International Conference on Compiler Construction*, Mar. 2002.
15. L. Semeria and G. D. Micheli. SpC: synthesis of pointers in C, application of pointer analysis to the behavioral synthesis from C. In *Proceedings of 21st IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 1998.
16. Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *Proceedings of the 32th Annual Symposium on Principles of Programming Languages (POPL)*, January 2005.