Scalable Error Detection using Boolean Satisfiability

Yichen Xie* Alex Aiken*

Computer Science Department
Stanford University
Stanford, CA 94305
{yxie,aiken}@cs.stanford.edu

ABSTRACT

We describe a software error-detection tool that exploits recent advances in boolean satisfiability (SAT) solvers. Our analysis is path sensitive, precise down to the bit level, and models pointers and heap data. Our approach is also highly scalable, which we achieve using two techniques. First, for each program function, several optimizations compress the size of the boolean formulas that model the control- and data-flow and the heap locations accessed by a function. Second, summaries in the spirit of type signatures are computed for each function, allowing inter-procedural analysis without a dramatic increase in the size of the boolean constraints to be solved.

We demonstrate the effectiveness of our approach by constructing a lock interface inference and checking tool. In an interprocedural analysis of more than 23,000 lock related functions in the latest Linux kernel, the checker generated 300 warnings, of which 179 were unique locking errors, a false positive rate of only 40%.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Experimentation, Languages, Verification.

Keywords

Program analysis, error detection, boolean satisfiability.

*Supported by NFS grant CCF-0430378.

POPL'05, January 12–14, 2005, Long Beach, California, USA.

1. INTRODUCTION

This paper presents Saturn[1], a software error-detection tool based on exploiting recent advances in solving boolean satisfiability (SAT) constraints. Rapid improvements in algorithms for SAT have led to its use in a variety of applications, including recently in program verification [16, 17, 19, 7, 11]. The contributions of our approach are:

- We give a translation from a realistic programming language to SAT that is substantially more efficient than previous approaches (Section 2). While we model each bit path sensitively as in [19, 8, 24], several techniques achieve a substantial reduction in the size of the SAT formulas Saturn must solve.
- We describe how to compute a summary, similar to a type signature, for each analyzed function (Section 4), thereby enabling interprocedural analysis. Computing concise summaries enables Saturn to analyze much larger programs than previous error checking systems based on SAT, and in fact, the scaling behavior of Saturn is at least competitive with, if not better than, other, non-SAT approaches to bug finding and verification. In addition, Saturn is able to infer and apply summaries that encode a form of interprocedural path sensitivity via input and output predicates, lending itself well to checking complex value-dependent program behaviors (Section 5.2).
- We present an interface for defining temporal safety properties for Saturn to check, and we show one such specification in detail: checking that a single thread correctly manages locks—i.e., does not perform two lock or unlock operations in a row on any lock (Section 5.5).
- We present a significant experiment in which we perform an interprocedural analysis of the entire Linux kernel for locking errors (Section 6). In analyzing 4.8 million lines of code, Saturn generated 300 warnings, of which 179 were unique locking errors, for a false positive rate of only 40% (i.e., more than half of the warnings were genuine bugs). For comparison, two previous studies of the same locking properties using MC [13] and CQual [12] found 31 and 18 locking errors respectively, with higher false positive rates (above 80% for both MC and CQual—i.e., fewer than 1 in 5 warnings is a genuine bug) [2, 10, 12].

[1]SATisfiability-based failURe aNalysis.

- We have built a database of all the inferred locking signatures of functions in the Linux kernel. We report some interesting statistics, showing that locking behavior in Linux is quite complex and that much of the kernel relies either directly or indirectly on correct locking behavior (Section 6.2).

It is worth noting that Saturn can do more than analyzing locks, although we do not report on additional checkers here. Locking analysis has become a standard test case for bug finding/verification tools, because in practice analyzing locking requires accurate modeling of many features: locking is always a flow sensitive and sometimes a path sensitive property, programmers store locks in data structures, pass locks as arguments, use complex tests to decide when and where to acquire and release locks, and so on.

One thing that Saturn is not, at least in its current form, is a verification tool. Tools such as CQual are capable of verification (proving the absence of bugs, or at least as close as one can reasonably come to that goal for C programs). Saturn is a bug finding tool in the spirit of MC, which means it is designed to find as many bugs as possible with a low false positive rate, potentially at the cost of missing some bugs. In particular, Saturn currently is unsound in its analysis of loops (see Section 2) because it is not possible in most cases to construct a finite boolean formula representing a fully unrolled loop. This problem is shared by all SAT-based analysis systems (see, e.g., [19, 24]) and we leave further investigation as future work.

Another source of unsoundness is Saturn's treatment of pointers in a function's environment. In order to reduce false positives, Saturn makes the assumption that distinct pointers from the environment do not alias each other. The same treatment is afforded to pointer values obtained from aspects of the C language that Saturn does not currently model (e.g., arrays, unions, inline assembly, unsafe casts, and function pointers). A sound alternative is to use a separate global alias analysis to account for potentially missed aliasing relationships. The treatment of external pointers is further discussed in Section 3.2.

2. THE SATURN FRAMEWORK

In this section, we present a low-level programming language and its translation into our error detection framework. Because our implementation targets C programs, our language models scalars, structures, pointers, and handles the arbitrary control flow[2] found in C. We begin with a language and translation that handles only scalar program values (Section 2.1), and gradually add features until we have presented the entire framework: arbitrary intraprocedural control flow including loops (Section 2.2), structures (Section 2.3), and finally pointers (Section 2.4). In Section 3 we consider some techniques that substantially improve the performance of our translation.

2.1 Scalar Translation

Figure 1 presents a grammar for a scalar language. The parenthesized symbol on the left hand side of each production is a variable ranging over elements of its syntactic category.

[2]The current implementation of Saturn handles reducible flow-graphs, which are by far the most common form even in C code. Irreducible flow-graphs can be converted to reducible ones by node-splitting [1].

Language

Type (τ) ::= $(n, \mathsf{signed} \mid \mathsf{unsigned})$

Obj (o) ::= v

Expr (e) ::= $\mathsf{unknown}(\tau) \mid \mathsf{const}(n, \tau) \mid o \mid \mathsf{unop}\ e \mid e_1\ \mathsf{binop}\ e_2 \mid (\tau)\ e \mid \mathsf{lift}_e(c, \tau)$

Cond (c) ::= $\mathsf{false} \mid \mathsf{true} \mid \neg\ c \mid e_1\ \mathsf{comp}\ e_2 \mid c_1 \wedge c_2 \mid c_1 \vee c_2 \mid \mathsf{lift}_c(e)$

Stmt (s) ::= $o \leftarrow e \mid \mathsf{assert}(c) \mid \mathsf{assume}(c) \mid \mathsf{skip}$

$\mathsf{unop} \in \{-, !\}$
$\mathsf{binop} \in \{+, -, *, /, \mathsf{mod}, \mathsf{band}, \mathsf{bor}, \mathsf{xor}, \ll, \gg_l, \gg_a\}$
$\mathsf{comp} \in \{=, >, \geq, <, \leq, \neq\}$

Representation

Rep (β) ::= $[b_{n-1} \ldots b_0]_s$ where $s \in \{\mathsf{signed}, \mathsf{unsigned}\}$
Bit (b) ::= $0 \mid 1 \mid x \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid \neg b$

Figure 1: A scalar language.

The language is statically and explicitly typed; the type rules are completely standard and for the most part we elide types for brevity. There are two base types: booleans (bool) and n-bit signed or unsigned integers (int). Note the base types are syntactically separated in the language as expressions, which are integer-valued, and conditions, which are boolean-valued. We use τ to range solely over different types of integer values.

The integer expressions include constants (const), integer variables (v), unary and binary operations, integer casts, and lifting from conditionals. We give the list of operators that we model precisely using boolean formulas (e.g. +, -, bitwise-and, etc.); for other operators (e.g., division, remainder, etc.), we make approximations. We use a special expression unknown to model unknown values (e.g., in the environment) and the result of operations that we do not model precisely.

Objects in the scalar language are n-bit signed or unsigned integers, where n and the signedness are determined by the type τ. As shown at the bottom of Figure 1, a separate boolean expression models each bit of an integer and thus tracking the width is important for our translation. The signed/unsigned distinction is needed to precisely model low-level type casts, bit shift operations, and arithmetic operations.

The class of Objects (Obj) ultimately includes variables, pointers, and structures, which encompass all the entities that can be the target of an assignment. For the moment we describe only scalar variables.

The translation for a representative selection of constructs is shown in Figure 2; the translations for the omitted cases introduce no new ideas. The translation rules for expressions

Expressions

$$\frac{\beta = \psi(v)}{\psi \vdash v \overset{\mathtt{E}}{\Rightarrow} \beta} \quad \text{scalar}$$

$$\frac{\begin{array}{c}(n,s) = \tau \\ x_0, \ldots, x_{n-1} \text{ are fresh boolean variables}\end{array}}{\psi \vdash \mathsf{unknown}(\tau) \overset{\mathtt{E}}{\Rightarrow} [x_{n-1} \ldots x_0]_s} \quad \text{unknown}$$

$$\frac{\psi \vdash e \overset{\mathtt{E}}{\Rightarrow} [b_{n-1} \ldots b_0]_x \quad \tau = (m,s) \quad b'_i = \begin{cases} b_i & \text{if } 0 \le i < n \\ 0 & \text{if } s = \mathsf{unsigned} \text{ and } n \le i < m \\ b_{n-1} & \text{if } s = \mathsf{signed} \quad \text{and } n \le i < m \end{cases}}{\psi \vdash (\tau)\ e \overset{\mathtt{E}}{\Rightarrow} [b'_{m-1} \ldots b'_0]_s} \quad \text{cast}$$

$$\frac{(n,s) = \tau \quad \psi \vdash c \overset{\mathtt{C}}{\Rightarrow} b}{\psi \vdash \mathsf{lift}_e(c,\tau) \overset{\mathtt{E}}{\Rightarrow} [\underbrace{00\cdots0}_{n-1} b]_s} \quad \mathsf{lift}_e$$

$$\frac{\begin{array}{c}\psi \vdash e \overset{\mathtt{E}}{\Rightarrow} [b_{n-1} \ldots b_0]_s \\ \psi \vdash e' \overset{\mathtt{E}}{\Rightarrow} [b'_{n-1} \ldots b'_0]_s\end{array}}{\psi \vdash e \ \mathtt{band}\ e' \overset{\mathtt{E}}{\Rightarrow} [b_{n-1} \wedge b'_{n-1} \ldots b_0 \wedge b'_0]_s} \quad \text{and}$$

Conditionals

$$\frac{\psi \vdash e \overset{\mathtt{E}}{\Rightarrow} [b_{n-1} \ldots b_0]_s}{\psi \vdash \mathsf{lift}_c(e) \overset{\mathtt{C}}{\Rightarrow} \bigvee_i b_i} \quad \mathsf{lift}_c$$

Statements

$$\frac{\psi \vdash e \overset{\mathtt{E}}{\Rightarrow} \beta}{\mathcal{G}, \psi \vdash (v \leftarrow e) \overset{\mathtt{S}}{\Rightarrow} \langle \mathcal{G}; \psi[v \mapsto \beta] \rangle} \quad \text{assign}$$

$$\frac{\psi \vdash c \overset{\mathtt{C}}{\Rightarrow} b}{\mathcal{G}, \psi \vdash \mathsf{assume}(c) \overset{\mathtt{S}}{\Rightarrow} \langle \mathcal{G} \wedge b; \psi \rangle} \quad \text{assume}$$

$$\frac{\psi \vdash c \overset{\mathtt{C}}{\Rightarrow} b \quad (\mathcal{G} \wedge \neg b) \text{ not satisfiable}}{\mathcal{G}, \psi \vdash \mathsf{assert}(c) \overset{\mathtt{S}}{\Rightarrow} \langle \mathcal{G}; \psi \rangle} \quad \text{assert-ok}$$

Figure 2: The translation.

$$\mathsf{MergeScalar}\left(v, \overline{(\mathcal{G}_i, \psi_i)}\right) = [b'_m \ldots b'_0]_s$$
$$\text{where } \begin{cases} [b_{im} \ldots b_{i0}]_s = \psi_i(v) \\ b'_j = \bigvee_i (\mathcal{G}_i \wedge b_{ij}) \end{cases}$$
$$\mathsf{MergeEnv}\left(\overline{(\mathcal{G}_i, \psi_i)}\right) = \langle \bigvee_i \mathcal{G}_i; \psi \rangle$$
$$\text{where } \psi(v) = \mathsf{MergeScalar}\left(v, \overline{(\mathcal{G}_i, \psi_i)}\right)$$

Figure 3: Merging control-flow paths.

have the form

$$\psi \vdash e \overset{\mathtt{E}}{\Rightarrow} \beta$$

which means that under the environment ψ mapping variables to vectors of boolean expressions (one for each bit in the variable's type), the expression e is translated to the vector of boolean expressions β.

The translation scheme for conditionals

$$\psi \vdash c \overset{\mathtt{C}}{\Rightarrow} b$$

is similar, except the target is a single boolean expression b modeling the condition. The most interesting rules are for statements, which have the form

$$\mathcal{G}, \psi \vdash s \overset{\mathtt{S}}{\Rightarrow} \langle \mathcal{G}'; \psi' \rangle$$

which means that under guard $\mathcal{G}$ and variable environment ψ the statement s results in a new guard/environment pair $\langle \mathcal{G}'; \psi' \rangle$. In our system, guards express path sensitivity; every statement is guarded by a boolean expression expressing the conditions under which that statement may execute. Most statements do not affect guards (the exception is `assume`); the important operations on guards are discussed in Section 2.2. A key statement in our language is `assert`, which we use to express points at which satisfiability queries must be checked. A statement `assert(c)` checks that $\neg$c cannot be true at that program point by computing the satisfiability of $\mathcal{G} \wedge \neg b$, where b is the translation of the condition c.

The overall effect of our translation is to perform straightforward symbolic execution, cast in terms of boolean expressions. Each statement transforms an environment into a new environment (and guard) that captures the effect of the statement. If all bits in the initial environment ψ_0 are concrete 0's and 1's and there are no unknown expressions in the program being analyzed, then in fact this translation is exactly symbolic execution and all modeled bits can themselves be reduced to 0's and 1's. However, bits may also be boolean variables (unknowns). Thus each bit b represented in our translation may be an arbitrary boolean expression over such variables.

2.2 Control Flow

We represent function bodies as control-flow graphs, which we define informally. Each statement s is a node in the graph, and each edge (s, s') represents an unconditional transfer of control from s to s'. If a statement has multiple successors, then execution may be transferred to any successor non-deterministically.

To model the deterministic semantics of conventional programs, we require that if a node has multiple successors, then each successor is an `assume` statement, and furthermore, that the conditions in those `assume`s are mutually exclusive and that their disjunction is equivalent to `true`. Thus a conditional branch with predicate p is modeled by a statement with two successors: one successor `assume`s p (the true branch) and the other `assume`s $\neg p$ (the false branch).

The other important issue is assigning a guard and environment to each statement s. Assume s has an ordered list of predecessors $\overline{s_i}$.[3] The translation of s_i produces an environment ψ_i and guard $\mathcal{G}_i$. The initial guard and environment for s is then a combination of the final guards

[3]We use the notation $\overline{X_i}$ as a shorthand for a vector of similar entities: $X_1 \ldots X_n$.

and environments of its predecessors. The desired guard is simply the disjunction of the predecessor guards; as we may arrive at s from any of the predecessors, s may be executed if any predecessor's guard is true. Note that due to the mutual exclusion assumption for branch conditions, at most one of the predecessor's guard can be true at a time. The desired environment is more complex, as we wish to preserve the path-sensitivity of our analysis down to the bit level. Thus, the value of each bit of each variable in the environment for each predecessor s_i of s must include the guard for s_i as well. This motivates the function MergeScalar in Figure 3, which implements a multiplexer circuit that selects the appropriate bits from the input environments ($\psi_i(v)$) based on the predecessor guards ($\mathcal{G}_i$). Finally, MergeEnv combines the two components together to define the initial environment and guard for s.

Preserving path sensitivity for every modeled bit is clearly expensive and it is easy to construct realistic examples where the number of modeled paths is exponential in the size of the control-flow graph. In Section 3.3 we present an optimization that enables us to make this approach work in practice.

Finally, every control-flow graph has a distinguished entry statement with no predecessors. The guard for this initial statement is `true`. We postpone discussion of the initial environment ψ_0 to Section 3.2 where we describe the lazy modeling of the external execution environment.

As mentioned in Section 1, we treat loops unsoundly. Loops are simply unrolled a number of times and the backedges removed from the control-flow graph. Thus, every function body is represented by an acyclic control-flow graph. While this handling of loops is unsound and a limitation of our current approach, we have found it to be effective in practice (see Section 6).

2.3 Structures

The program syntax and translation of structures is given in Figure 4. A structure is a data type with named fields, which we represent as a set of (*field_name*, *object*) pairs. We extend the syntax of types (resp. objects) with sets of types (resp. objects) labeled by field names, and similarly the representation of a `struct` in C is the representation of the fields also labeled by the field names. The shorthand notation $o.f_i$ selects the object of field f_i from object o.

The function RecAssign does the work of structure assignment. As expected, assignment of structures is defined in terms of assignments of its fields. Because structures may themselves be fields of structures, RecAssign is recursively defined.

2.4 Pointers

The final and technically most involved construct in our translation is pointers. The basic idea is to maintain path sensitivity for heap locations, as we do for variables in the environment. As with variables, we must allow for the pointer to have different values (point to different locations) depending on the execution history (the guard). The language definition and translation rules are given in Figure 5 and 6. Pointers point to *locations*, which are either an object o or the constant `null` (i.e., a null pointer). A *guarded location* is a pair of a guard and a location. A *guarded location set (GLS)*, which we write as $\{\!|\ \ldots\ |\!\}$ to distinguish from other

Language

$$\text{Type } (\tau) ::= \{(f_1,\tau_1),\ldots,(f_n,\tau_n)\} \mid \ldots$$
$$\text{Obj } (o) ::= \{(f_1,o_1),\ldots,(f_n,o_n)\} \mid \ldots$$

Shorthand

$$\frac{o = \{(f_1,o_1),\ldots,(f_n,o_n)\}}{o.f_i \stackrel{\text{def}}{=} o_i} \quad \text{field-access}$$

Representation

$$\text{Rep } (\beta) ::= \{(f_1,\beta_1),\ldots,(f_n,\beta_n)\} \mid \ldots$$

Translation

$$\frac{o = \{(f_1,o_1),\ldots,(f_n,o_n)\} \quad \psi \vdash o_i \stackrel{\text{E}}{\Rightarrow} \beta_i \quad \text{for } i \in 1..n}{\psi \vdash o \stackrel{\text{E}}{\Rightarrow} \{(f_1,\beta_1),\ldots,(f_n,\beta_n)\}} \quad \text{object-str}$$

$$\mathsf{RecAssign}(\psi, v, \beta) = \psi[v \mapsto \beta]$$
$$\mathsf{RecAssign}(\psi, o, \beta) = \psi_n$$
$$\text{where } \begin{cases} o = \{(f_1,o_1),\ldots,(f_n,o_n)\} \\ \beta = \{(f_1,\beta_1),\ldots,(f_n,\beta_n)\} \\ \psi_0 = \psi \\ \psi_i = \mathsf{RecAssign}(\psi_{i-1}, o_i, \beta_i)\ (\forall\ i \in 1..n) \end{cases}$$

$$\frac{\psi \vdash e \stackrel{\text{E}}{\Rightarrow} \beta \quad \psi' = \mathsf{RecAssign}(\psi, o, \beta)}{\mathcal{G}, \psi \vdash (o \leftarrow e) \stackrel{\text{S}}{\Rightarrow} \langle \mathcal{G}; \psi' \rangle} \quad \text{assign-struct}$$

Figure 4: Translation of structures.

kinds of sets, is a set of guarded locations. We define

$$\{\!|\ \ldots,(\mathcal{G},l),(\mathcal{G}',l),\ldots\ |\!\} \equiv \{\!|\ \ldots,(\mathcal{G}\vee\mathcal{G}',l),\ldots\ |\!\}$$

and we assume that a GLS is normalized using this equivalence so that a location occurs at most once in the set. We adopt the convention that the guard for `null` is listed first with guard $\mathcal{G}_0$.

We briefly explain the rules in Figure 5. As shown in the rule pointer, the mapping from pointers to GLS's is part of the environment. Taking the address of an object (rule getaddr-obj) simply creates a GLS with a single entry—the object itself with no guard (the guard `true`). Taking the address of a series of field dereferences from a pointer means taking each location the pointer could point to and creating a guarded location from the object reached by performing the field dereferences.[4] The rule lift_c-pointer expresses that a pointer treated as a condition is true if it is non-null. The rule malloc creates a fresh object that is the target of the malloc'd pointer. Note that malloc is silent on the contents of the new object; there are different choices (all bits zero, all bits unconstrained) depending on the analysis application and what one assumes about the memory allocator. The rule store models an indirect assignment through a pointer,

[4]The treatment of `null` here is needed to be consistent with the ANSI C standard.

Language

$$\begin{array}{rl} \text{Type} & (\tau) ::= \tau \text{ pointer} \mid \ldots \\ \text{Obj} & (o) ::= p \mid \ldots \\ \text{Deref} & (m) ::= p \rightarrow f_1.\cdots.f_n \\ \text{Expr} & (e) ::= \text{null} \mid \&o \mid \&m \mid \ldots \\ \text{Stmt} & (s) ::= \text{load}(m,o) \mid \text{store}(m,e) \mid \text{malloc}(p) \mid \ldots \end{array}$$

Representation

$$\begin{array}{rl} \text{Loc} & (l) ::= \text{null} \mid o \\ \text{Rep} & (\beta) ::= \{\!|\ (\mathcal{G}_1, l_1), \ldots, (\mathcal{G}_k, l_k)\ |\!\} \mid \ldots \end{array}$$

Translation

$$\frac{\beta = \psi(p)}{\psi \vdash p \overset{\text{E}}{\Rightarrow} \beta} \quad \text{pointer}$$

$$\frac{}{\psi \vdash \&o \overset{\text{E}}{\Rightarrow} \{\!|\ (\text{true}, o)\ |\!\}} \quad \text{getaddr-obj}$$

$$\frac{\begin{array}{l} m = p \rightarrow f_1.\cdots.f_n \\ \psi \vdash p \overset{\text{E}}{\Rightarrow} \{\!|\ (\mathcal{G}_0, \text{null}), \overline{(\mathcal{G}_i, o_i)}\ |\!\} \\ \beta = \{\!|\ (\mathcal{G}_0, \text{null}), \overline{(\mathcal{G}_i, o_i.f_1.\cdots.f_n)}\ |\!\} \end{array}}{\psi \vdash \&m \overset{\text{E}}{\Rightarrow} \beta} \quad \text{getaddr-mem}$$

$$\frac{\psi(p) = \{\!|\ (\mathcal{G}_0, \text{null}), \overline{(\mathcal{G}_i, o_i)}\ |\!\}}{\psi \vdash \text{lift}_c(p) \overset{\text{C}}{\Rightarrow} \bigvee_{i \neq 0} \mathcal{G}_i} \quad \text{lift}_c\text{-pointer}$$

$$\frac{\beta = \{\!|\ (\text{true}, o)\ |\!\} \quad (o \text{ fresh})}{\mathcal{G}, \psi \vdash \text{malloc}(p) \overset{\text{S}}{\Rightarrow} \langle \mathcal{G}; \psi[p \mapsto \beta] \rangle} \quad \text{malloc}$$

$$\frac{\begin{array}{l} m = p \rightarrow f_1.\cdots.f_n \\ \psi \vdash p \overset{\text{E}}{\Rightarrow} \{\!|\ (\mathcal{G}_0, \text{null}), (\mathcal{G}_1, o_1), \ldots, (\mathcal{G}_k, o_k)\ |\!\} \\ \mathcal{G}' = \mathcal{G} \wedge \neg \mathcal{G}_0 \\ \mathcal{G}' \wedge \mathcal{G}_i, \psi \vdash (o_i.f_1.\cdots.f_n \leftarrow e) \overset{\text{S}}{\Rightarrow} \langle \mathcal{G}_i; \psi_i \rangle \\ \qquad\qquad\qquad\qquad\qquad\qquad (\text{for } i \in 1..k) \end{array}}{\mathcal{G}, \psi \vdash \text{store}(m,e) \overset{\text{S}}{\Rightarrow} \text{MergeEnv}\left(\overline{(\mathcal{G}_i; \psi_i)}\right)} \quad \text{store}$$

Figure 5: Pointers and guarded location sets.

Merging

$$\text{AddGuard}\,(\mathcal{G}, \{\!|\ (\mathcal{G}_1, l_1), .., (\mathcal{G}_k, l_k)\ |\!\}) = \{\!|\ (\mathcal{G} \wedge \mathcal{G}_1, l_1), .., (\mathcal{G} \wedge \mathcal{G}_k, l_k)\ |\!\}$$

$$\text{MergePointer}\left(p, \overline{(\mathcal{G}_i, \psi_i)}\right) = \bigcup_i \text{AddGuard}(\mathcal{G}_i, \psi_i(p))$$

$$\text{MergeEnv}\left(\overline{(\mathcal{G}_i, \psi_i)}\right) = \langle \textstyle\bigvee_i \mathcal{G}_i; \psi \rangle$$

$$\text{where} \begin{cases} \psi(v) = \text{MergeScalar}\left(v, \overline{(\mathcal{G}_i, \psi_i)}\right) \\ \psi(p) = \text{MergePointer}\left(p, \overline{(\mathcal{G}_i, \psi_i)}\right) \end{cases}$$

Figure 6: Control-flow merges with pointers.

possibly involving field dereferences, by combining the results for each possible location the pointer could point to. The line $\mathcal{G}' = \mathcal{G} \wedge \neg\mathcal{G}_0$ adds an assumption that the pointer is not `null`; if we are interested in null-pointer checks that can be done by a separate null-pointer checker built on our framework. We omit the rule for load, as it is very similar to store.

3. DISCUSSION AND IMPROVEMENTS

In this section, we discuss how our translation reduces the size of satisfiability queries by achieving a form of program slicing. We also discuss two improvements to our approach. The first (Section 3.2) concerns how we treat inputs of unknown shape to functions and the second (Section 3.3) is an optimization that greatly reduces the cost of guards.

3.1 Automatic Slicing

Program slicing is a technique to simplify a program by removing the parts that are irrelevant to the property we are interested in. Slicing is commonly done by computing control and data dependencies and preserving only the statements that the property depends on. We show that our translation automatically slices a program and only uses clauses that the current SAT query requires.

Consider the following program snippet below:

```
if (x) y = a; else y = b;
z = /* complex computation here */;
if (z) ... else ...;
assert(y < 5);
```

The computation of z is irrelevant to the property we are checking ($y < 5$). The variable y is data dependent on a and b and control dependent on x. Using the translation rules in Section 2, we see that the translation of $y < 5$ only involves the bits in x, a, and b, but not z, because the assign rule accounts for the data dependencies and the merge rule pulls in the control dependency, and nothing else is included. Because properties of interest often depend on a small portion of the program, this design helps keep the size of SAT queries under control.

3.2 Lazy Construction of the Environment

A standard problem in modular program analysis systems is the modeling of the external environment. In particular, we need a method to model and track data structures used, but not created by the code fragment being analyzed.

There is no consensus on the best solution to this problem. To the best of our knowledge, SLAM [4] and Blast [15] require manual construction of the environment. For example, to analyze a module that manipulates a linked list of locks defined elsewhere, these systems likely require a harness that populates an input list with locks. The problem is reduced as the target code-bases (e.g., Windows drivers in the case for SLAM) can often share a carefully crafted harness (e.g., a model for the Windows kernel) [3]. Nevertheless, the need to "close" the environment represents a substantial manual effort in the deployment of such systems.

Because we achieve scalability by computing function summaries, we must analyze a function independent of its calling context and still model its arguments. Our solution is similar in spirit to the lazy initialization algorithm described in [18]. Recall in Section 2, values of variables referenced but not created in the code, i.e., those from the external environment, are defined in the initial evaluation environment

ψ_0. Saturn lazily constructs ψ_0 by calling a special function DefVal, which is supplied by the analysis designer and maps all external objects to a checker-specific estimation of their default values. ψ_0 is then defined as DefVal(v) for all v. Operationally, DefVal is applied lazily, only when uninitialized objects are encountered during symbolic evaluation. This allows us to model potentially unbounded data structures in the environment. Besides its role in defining the initial environment ψ_0, DefVal is also used to provide an estimation of the return values and side-effects of function calls (Section 5.3).

In our implementation, we model scalars from the environment with a vector of unconstrained boolean variables. For pointers, we use the common assumption that distinct pointers from the environment do not alias each other. This can be modeled by a DefVal that returns a fresh location for each unknown pointer dereference.[5] A sound alternative would be to use a separate global alias analysis as part of the definition of ψ_0. Note once a pointer is initialized, Saturn performs an accurate intraprocedural path-sensitive analysis, including respecting alias relationships, on that pointer.

3.3 Using BDDs for Guards

Consider the following high-level code fragment:

```
if c then ... else ... ; s
```

After translation to a control-flow graph, there are two paths reaching the statement `s` with guards `c` and $\neg$`c`. Thus the guard of `s` is `c` $\vee \neg$`c`. Since guards are attached to every bit of every modeled location at every program point, it is important to avoid growth in the size of guards at every control-flow merge. One way to accomplish this task is to decompile the unrolled control flow graph into structured programs with only `if` statements, so that we know exactly where the branch conditionals cancel. However, this approach requires code duplication in the presence of `goto`, `break`, and `continue` statements commonly found in C.

Our solution is to introduce an intermediate representation of guards using binary decision diagrams [5]. We give each condition (which may be a complex expression) a name and use a BDD to represent the boolean combination of all conditions that enable a program path. At control-flow merges we join the corresponding BDDs. The BDD join operation can simplify the representation of the boolean formula to a canonical form; for example, the join of the BDDs for `c` and $\neg$`c` is represented by `true`. In our translation of a statement, we convert the BDD representing the set of conditions at that program point to the appropriate guard.

The simplification of guards also eliminates trivial control dependencies in the automatic slicing scheme described in Section 3.1. In the small example in that section, had we not simplified guards, the assertion would have been checked under the guard $(x \vee \neg x) \wedge (z \vee \neg z)$, which pulls in the otherwise irrelevant computation of z.

[5]In the implementation, DefVal(p) returns $\{\!|\ (\mathcal{G}, \mathsf{null}), (\neg\mathcal{G}, o)\ |\!\}$, where $\mathcal{G}$ is an unconstrained bit variable, and o is a fresh object of the appropriate type. This allows us to model common data structures like linked lists and trees of arbitrary length or depth. A slightly smarter variant handles doubly linked lists and trees with parent pointers knowing one node in such a data structure.

4. INTERPROCEDURAL ANALYSIS

The misspecification and misunderstanding of function interface constraints is a major source of errors in large software systems. To detect such errors, we must perform interprocedural analysis.

There are two well known approaches to interprocedural analysis: function inlining and compositional analysis based on summaries. Inlining is a simple technique used by a number of whole program analysis algorithms to analyze beyond function boundaries (e.g., [19, 14]). However, it discards natural boundaries in software systems and may lead to exponential explosion in code size. Therefore, this approach does not readily scale to the size of systems we aim to check.

In a summary-based approach, each function or module is abstracted into a concise representation which summarizes the observable behavior of the function with regard to some property. The benefit is two fold: First, the summary for a particular function can be computed once and used at all call sites, thereby avoiding redundant analysis. Second, a function summary is usually expressed in terms of an abstraction of the observable behavior of a function, thereby hiding irrelevant details of the function and simplifying the analysis of its callers without losing relevant information. However, finding the right abstraction is crucial, especially for bug detection systems where being overly conservative translates into large false positive rates, greatly reducing the usefulness of the tool.

We provide a query-response interface that allows a user-defined checker to compute function summaries by posing satisfiability queries about program execution. In the following section, we give an inference and checking framework for finite state properties, also known as temporal safety properties, using Saturn.

5. CHECKING FINITE STATE PROPERTIES

Finite state properties are a class of specifications that can be described as certain program values passing through a finite set of states, over time, under specific conditions. Locking, where a lock can legally only go from the unlocked state to the locked state and then back to the unlocked state, is a canonical example. These properties are also referred to as temporal safety properties.

In this section, we focus on finite state properties, and describe a summary based interprocedural analysis that uses the Saturn framework to automatically check such properties. We start by defining a common name space for shared objects between the caller and the callee (Section 5.1), which we use to define a general summary representation for finite state properties (Section 5.2). We then describe algorithms for applying (Section 5.3) and inferring (Section 5.4) function summaries in the Saturn framework. We conclude by describing our implementation of an interprocedural lock checker (Section 5.5).

5.1 Interface Objects

In C, the two sides of a function invocation share the global name space, but have separate local name spaces. Thus we need a common name space for objects referred to in the summary. Barring external channels and unsafe memory accesses, the two parties share values through global variables and parameters. Therefore, shared objects can be named using a path from one of these two roots.

We formalize this idea using *interface objects* (IObj) as common names for objects shared between caller and callee:

$$\mathsf{IObj}\ (l) ::= \mathsf{param}_i \mid \mathsf{global\_var} \mid \mathsf{ret\_val} \mid\ *l \mid l.f$$

Dependencies across function calls are expressed by interface expressions (IExpr) and conditions (ICond), which are defined respectively by replacing references to objects with interface objects in the definition of Expr and Cond (as defined in Figure 1, and extended in Figure 5).

To perform interprocedural analysis of a function, we must map input interface objects to the names used in the function body, perform symbolic evaluation of the function, and map the final function state to the final state of the interface objects. Thus, we need two mappings to convert between interface objects and those in the native name space of a function:

$$[\![\cdot]\!]_{\mathsf{args}} : \mathsf{IObj} \rightarrow \mathsf{Obj}^{\mathsf{ext}} \text{ and } [\![\cdot]\!]^{-1}_{\mathsf{args}} : \mathsf{Obj} \rightarrow \mathsf{IObj}$$

Converting IObj's to native objects is straightforward. For function call $r = f(a_0, ..., a_n)$,

$$\begin{aligned} [\![\mathsf{global}]\!]_{a_0 \ldots a_n} &= \mathsf{global} \\ [\![\mathsf{param}_i]\!]_{a_0 \ldots a_n} &= a_i \\ [\![\mathsf{ret\_val}]\!]_{a_0 \ldots a_n} &= r \\ [\![*l]\!]_{a_0 \ldots a_n} &= *([\![l]\!]_{a_0 \ldots a_n}) \\ [\![l.f]\!]_{a_0 \ldots a_n} &= ([\![l]\!]_{a_0 \ldots a_n}).f \end{aligned}$$

The range of the conversion is $\mathsf{Obj}^{\mathsf{ext}}$, which allows dereferencing of pointers inside the expression. This feature is not in the language defined in Section 2, but can be eliminated by using temporary variables and explicit load/store operations.

The inverse conversion is more involved, since there may be multiple aliases of the same object in the program. We incrementally construct the $[\![\cdot]\!]^{-1}_{\mathsf{args}}$ mapping for objects accessed through global variables and parameters. For example, in

```
void f(struct str *p) { spin_lock(&p->lock); }
```

the corresponding interface object for p is param_0, since it is defined as the first formal parameter of f. Recall that the object pointed to by p→lock is lazily instantiated when p is dereferenced by calling DefVal(p) (see Section 3). As part of the instantiation, we initialize every field of the struct (*p), and compute the appropriate IObj for each field at that time. Specifically, the interface object for p→lock is $(*\mathsf{param}_0).\mathsf{lock}$.

The conversion operations extend to interface expressions and conditionals. For brevity, name space conversions for objects, expressions, and conditionals are mostly kept implicit in the discussion below.

5.2 Function Summary Representation

The language for expressing finite state summaries is given in Figure 7. Each function summary is a four-tuple consisting of:

- a set of input predicates P_{in},
- a set of output predicates P_{out},
- a set of interface objects M, which may be modified during the function call, and
- a relation R summarizing the FSM behavior of the function.

$$\begin{aligned} &\mathsf{FSM\ States}\ \mathcal{S} = \{\mathsf{Error}, s_1, \ldots, s_n\} \\ &\mathsf{Summaries}\ \ \Sigma = \langle P_{in}, P_{out}, M, R \rangle \\ &\qquad \text{where}\ \ P_{in} = \{p_1, \ldots, p_n\} \quad p_i \in \mathsf{ICond}, \\ &\qquad\qquad P_{out} = \{q_1, \ldots, q_n\} \quad q_i \in \mathsf{ICond}, \\ &\qquad\qquad M \subseteq \mathsf{IObj}, \text{ and} \\ &\qquad\qquad R \subseteq \mathsf{IObj} \times 2^{|P_{in}|} \times \mathcal{S} \times 2^{|P_{out}|} \times \mathcal{S} \end{aligned}$$

Figure 7: Function summary representation.

The checker need only supply the set of FSM states and the input and output predicates; both M and R are computed automatically for each function by Saturn (see Section 5.4).

The FSM behavior of a function call is modeled as a set of state transitions of one or more interface objects. These transitions map input states to output states based on the values of a set of input (P_{in}) and output predicates (P_{out}). The state transitions are given in the set R. Each element in R is a five tuple: $(\mathsf{sm}, \mathsf{incond}, s, \mathsf{outcond}, s')$, which we describe below:

- $\mathsf{sm} \in \mathsf{IObj}$ is the object whose state is affected by the transition relationship. In the lock checker, sm identifies the accessed lock objects, as a function may access more than one lock during its execution.
- $\mathsf{incond} \in 2^{|P_{in}|}$ is used to denote the input condition $(\bigwedge_{i \in \mathsf{incond}} p_i) \wedge (\bigwedge_{i \notin \mathsf{incond}} \neg p_i)$ where $P_{in} = \{p_1, \ldots, p_n\}$. The value of incond is evaluated on entry to the function.
- $s \in S$ is the initial state of sm in the state transition.
- $\mathsf{outcond} \in 2^{|P_{out}|}$ is defined like incond and denotes the output condition of the transition. outcond is evaluated on exit.
- $s' \in S$ is the state of sm after the transition.

Figure 8 presents the summary of three sample locking functions: spin_lock, spin_trylock, and complex_wrapper. The function complex_wrapper captures some of the more complicated locking behavior in Linux. Nevertheless, we are able to express its behavior using our summary representation. We describe how function summaries are inferred and used in the following subsections.

5.3 Summary Application

This subsection describes how the summary of a function is used to model its behavior at a call site. For a given function invocation $f(a_0, \ldots, a_n)$, we translate the call into a set of statements simulating the observable effects of the function. The translation, given in Figure 9, is composed of two stages:

1. In the first stage, we save the values of relevant program states before and after the call (line 3-4 and 8 in Figure 9), and account for the side effects of the function by conservatively assigning unknown values to objects in the modified set M (line 6). Relevant values before the call include all input predicates p_i, and the current states (sm_i) of the interface objects mentioned in the transition relation R. Relevant values after the call include all output states q_i. We then

```
void complex_wrapper(spinlock_t *l, int flag, int *success)
{
    if (flag) *success = spin_trylock(l);
    else { spin_unlock(l); *success = 1; }
}
```

States: $S = \{\mathsf{Error} = 0, \mathsf{Locked} = 1, \mathsf{Unlocked} = 2\}$
Summary: $\Sigma = \langle M, R, P_{in}, P_{out} \rangle$

spin_lock :
Input: $P_{in} = \{\}$ $P_{out} = \{\}$
Output: $M = \{*\mathsf{param}_0\}$
$R = \{ (*\mathsf{param}_0, \_, \mathsf{Unlocked}, \_, \mathsf{Locked}),$
$(*\mathsf{param}_0, \_, \mathsf{Locked}, \_, \mathsf{Error})\}$

spin_trylock :
Input: $P_{in} = \{\}$ $P_{out} = \{\mathsf{lift}_c(\mathsf{ret\_val})\}$
Output: $M = \{*\mathsf{param}_0, \mathsf{ret\_val}\}$
$R = \{ (*\mathsf{param}_0, \_, \mathsf{Unlocked}, \mathsf{true}, \mathsf{Locked}),$
$(*\mathsf{param}_0, \_, \mathsf{Unlocked}, \mathsf{false}, \mathsf{Unlocked}),$
$(*\mathsf{param}_0, \_, \mathsf{Locked}, \_, \mathsf{Error})\}$

complex_wrapper :
Input: $P_{in} = \{\mathsf{lift}_c(\mathsf{param}_1)\}$ $P_{out} = \{\mathsf{lift}_c(*\mathsf{param}_2)\}$
Output: $M = \{*\mathsf{param}_0, *\mathsf{param}_2\}$
$R = \{ (*\mathsf{param}_0, \mathsf{true}, \mathsf{Locked}, \_, \mathsf{Error})$
$(*\mathsf{param}_0, \mathsf{true}, \mathsf{Unlocked}, \mathsf{true}, \mathsf{Locked})$
$(*\mathsf{param}_0, \mathsf{true}, \mathsf{Unlocked}, \mathsf{false}, \mathsf{Unlocked})$
$(*\mathsf{param}_0, \mathsf{false}, \mathsf{Unlocked}, \mathsf{true}, \mathsf{Error})$
$(*\mathsf{param}_0, \mathsf{false}, \mathsf{Locked}, \mathsf{true}, \mathsf{Unlocked})\}$

Figure 8: Sample function summaries for the locking property.

use an assume statement to rule out impossible combinations of input predicates and output predicates (line 10; e.g., some functions always return a non-NULL pointer).

2. In the second stage, we process the state transitions in R by first testing their activation conditions, and when satisfied, carrying out the transitions (line 14-16). incond denotes the condition $(\bigwedge_{i \in \mathsf{incond}} \widehat{p}_i) \wedge (\bigwedge_{i \notin \mathsf{incond}} \neg \widehat{p}_i)$; and the condition for outcond is symmetric. Notice that since incond and outcond are a valuation of all input and output predicates, no two transitions on the same state machine should be enabled simultaneously (we flag such cases as errors, since the caller would have no way of knowing the exit state of the function).

There is one aspect of the translation that is left unspecified in the description, which is the unknown values used to model the side-effects of the function call. For scalar values, we use the translation rule for unknown and conservatively model these values with a set of unconstrained boolean variables. For pointers, we extend the DefVal operator described in Section 3.2 to obtain a checker-specified estimation of the shape of the object being pointed to.

5.4 Summary Generation

This subsection describes how we compute the summary of a function after analysis. Before we proceed, we first state two assumptions about the translation from C to Saturn:

Assumption

$$\Sigma(f) = \langle P_{in}, P_{out}, M, R \rangle$$
$$\text{where } \begin{cases} P_{in} = \{p_1, \ldots, p_m\} \\ P_{out} = \{q_1, \ldots, q_n\} \\ M = \{o_1, \ldots, o_k\} \\ R = \{(\mathsf{sm}_1, \mathsf{incond}_1, s_1, \mathsf{outcond}_1, s_1'), \ldots, \\ \qquad (\mathsf{sm}_l, \mathsf{incond}_l, s_l, \mathsf{outcond}_l, s_l')\} \end{cases}$$

Instrumentation

```
1: (* Stage 1: Preparation *)
2: (* save useful program states *)
3: p̂1 ← p1; ...; p̂n ← pm;
4: sm̂1 ← sm1; ...; sm̂l ← sml;
5: (* account for the side-effects of f *)
6: o1 ← unknown(τo1); ...; ok ← unknown(τok);
7: (* save the values of output predicates *)
8: q1' ← q1; ...; qn' ← qn;
9: (* rule out infeasible comb. of incond and outcond *)
10: assume(⋁i(smi = si ∧ incondi ∧ outcondi));
11:
12: (* Stage 2: Transitions *)
13: (* record state transitions after the function call *)
14: if (sm̂1 = s1 ∧ incond1 ∧ outcond1) sm1 ← s1';
15:    ...
16: if (sm̂l = sl ∧ incondl ∧ outcondl) sml ← sl';
```

Figure 9: Summary application.

$$P_{in} = \{p_1, \ldots, p_m\}$$
$$P_{out} = \{q_1, \ldots, q_n\}$$
$$M = \{v \mid \mathsf{is\_satisfiable}(\psi_0(v) \neq \psi(v))\}$$
$$R = \{ (\mathsf{sm}, \mathsf{incond}, s, \mathsf{outcond}, s') \mid \mathsf{is\_satisfiable}(\psi_0(\mathsf{sm} = s) \wedge \psi_0(\mathsf{incond}) \wedge \psi(\mathsf{outcond}) \wedge \psi(\mathsf{sm} = s')) \}$$

Figure 10: Summary generation.

1. We assume that each function has one unique exit block. In case the function has multiple return statements, we add a dummy exit block linked to all return sites. The exit block is analyzed last (see Section 2) and the environment ψ at that point encodes all paths from function entry to exit. Summary generation is carried out after analyzing the exit block.

2. We model return statements in C by assigning the return value to a special object rv, and $[\![\mathsf{rv}]\!]^{-1}_{\mathsf{args}} = \mathsf{ret\_val}$.

Figure 10 gives the summary generation algorithm. The input is a set of input (P_{in}) and output predicates (P_{out}). The algorithm involves a series of queries to the SAT solver based on the initial (ψ_0) and final state (ψ) to determine: (1) the set of modified objects M, and (2) the set of transition relationships R. In computing M and R, we use a shorthand $\psi(x)$ to denote the valuation of x under environment ψ.

The summary generation algorithm proceeds as follows. Intuitively, modified objects are those whose valuation may be different under the initial environment ψ_0 and the final

environment ψ. We compute M by iterating over all interface objects v and use the SAT solver to determine whether the values may be different or not.

R is computed by enumerating over all relevant interface objects (e.g., locks in the lock checker) in the function and all combinations of input predicates and output predicates. We again use the SAT solver to determine whether a transition under a particular set of input and output predicates is feasible.

As one may have noticed, this process involves many SAT queries. We observe that if done carefully, each SAT query can be separated into two parts: (1) constraints that encode the program control and data flow; and (2) the specific valuations of the state variables and the input and output predicates. Part (1) is shared among all SAT queries. Incremental SAT solvers are able to share and reuse information learned (e.g., using conflict clauses) in the common part of the queries to speed up the SAT solving process. In practice, SAT queries typically complete in under one second.

5.5 A Linux Lock Checker

In this section, we use the FSM checking framework described above to construct a lock checker for the Linux kernel. We start with some background information, and list the challenges we encountered in trying to detect locking bugs in Linux. We then describe the lock checker we have implemented in the Saturn framework.

The Linux kernel is a widely deployed and well-tested core of the Linux operating system. The kernel is designed to scale to an array of multiprocessor platforms, and thus is inherently concurrent. It uses a variety of locking mechanisms (e.g., spin locks, semaphores, read/write locks, primitive compare and swap instructions, etc.) to coordinate concurrent accesses of kernel data structures. For efficiency reasons, most of the code in the kernel runs in the supervisor mode, and synchronization bugs can thus cause crashes or hangs that result in data losses and system down time. For this reason, locking bugs have received the attention of a number of research and commercial checking and verification efforts.

Locks (a.k.a. mutexes) are naturally expressed as a finite state property with three states: Locked, Unlocked, and Error. The lock operation can be modeled as two transitions: from Unlocked to Locked, and Locked to Error (unlock is similar). There are a few challenges that a checker must deal with to model locking behavior in Linux:

- **Aliasing**. In Linux, locks are passed by reference (i.e., by pointers in C). One immediate problem is the need to deal with pointer aliasing. CQual employs a number of techniques to infer non-aliasing relationships to help refine the results from the alias analysis [2]. MC [13] assumes non-aliasing among all pointers, which helps reduce false positives, but also limits the checking power of the tool.

- **Heap Objects**. In fine grained locking, locks are often embedded in heap objects. These objects are stored in the heap and passed around by reference. To detect bugs involving heap objects, a reasonable model of the heap needs to be constructed (recall Section 3.2). The need to write "drivers" that construct the checking environment has proven to be a non-trivial task in traditional model checkers [3].

- **Path Sensitivity**. The state machine for locks becomes more complex when we consider *trylocks*. Trylocks are lock operations that can fail. The caller must check the return value of trylocks to determine whether the operation has succeeded or not. Besides trylocks, some functions intentionally exit with locks held on error paths and expect their callers to carry out error recovery and cleanup work. These constructs are used extensively in Linux. In addition to that, one common usage scenario in Linux is the following:

  if (x) spin_lock(&l); ...; **if** (x) spin_unlock(&l);

  Some form of path sensitivity is necessary to handle these cases.

- **Interprocedural Analysis**. As we show in Section 6, a large portion of synchronization errors arise from misunderstanding of function interface constraints. The presence of more than 600 lock/unlock/trylock wrappers further complicates the analysis. Imprecision in the intraprocedural analysis is amplified in the interprocedural phase, so we believe a precise interprocedural analysis is important in the construction of a lock checker.

Our lock checker is based on the framework described above (see Figure 8). States are defined as usual: {Locked, Unlocked, Error}. To accurately model trylocks, we define $P_{out} = \{\mathsf{lift}_c(\mathsf{ret\_val})\}$ for functions that return integers or pointers. Tracking this predicate in summaries is also adequate for modeling functions that exit in different lock states depending on whether the return value is 0 (null) or not. P_{out} is defined to be the empty set for functions of type void. P_{in} is defined to be the empty set.

We detect two types of locking errors in Linux:

- **Type A: double locking/unlocking**. These are functions that may acquire or release the same lock twice in a row. The summary relationship R of such functions contains two transitions on the same lock: one leads from the Locked state to Error, and the other from the Unlocked state to Error. This signals an internal inconsistency in the function—no matter what state the lock is in on entry to the function, there is a path leading to the error state.

- **Type B: ambiguous return state**. These are functions that may exit in both Locked and Unlocked state with no observable difference (w.r.t. P_{out}, which is $\mathsf{lift}_c(\mathsf{ret\_val})$) in the return value. These bugs are commonly caused by missed operations to restore lock states on error paths.

6. EXPERIMENTAL RESULTS

Our implementation of Saturn is written in O'Caml [20] and makes use of several existing software packages. We use a modified version of the GNU C preprocessor to preserve a number of primitive lock operations that would otherwise be macro expanded into inline assembly instructions. Several "panic" primitives (e.g., `panic, BUG`) are also preserved in this phase. The preprocessed code then goes through a modified version of CIL [23], where it is parsed and translated into the Saturn modeling language described in Section 2. We use a GNU DBM (GDBM) database to index

Type	Count
Num. of Files	12455
Total Line Count	4.8 million LOC
Total Num. Func.	63850
Lock Related Func.	23458
Running time	19h40m CPU time
Approx. LOC/sec	67

Table 1: Performance statistics.

and store the processed function bodies to avoid redundant parsing and transformation. A separate GDBM database is also used to store function summaries as they are computed. During the analysis, we use the BuDDy [21] BDD package to track and simplify CFG block guards. Finally, we use zChaff [25, 22] as our backend SAT solver because of its performance and incremental SAT solving capabilities.

We have implemented the lock checker described in Section 5.5 as a plugin to the Saturn framework. The checker models locks in Linux (e.g., objects of type `spinlock_t, rwlock_t, rw_semaphore`, and `semaphore`) using the state machines defined in Section 5. When analyzing a function, we retrieve the lock summaries of its callees and use the algorithm described in Section 5.3 to simulate their observable effects. At the end of the analysis, we compute a summary for the current function using the algorithm described in Section 5.4 and store it in the summary database for future use.

The order of analysis for functions in Linux is determined by topologically sorting the static call graph of the Linux kernel. Recursive function calls are represented by strongly connected components (SCC) in the call graph. During the bottom up analysis, functions in SCCs are analyzed in arbitrary order.

We start the analysis by seeding the lock summary database with manual specifications of around 40 lock, unlock and trylock primitives in Linux. Otherwise the checking process is fully automatic: our tool works on the unmodified source tree and requires no human guidance during the analysis.

We ran our lock checker on the then latest release of the kernel source tree (v2.6.5). Performance statistics of the experiment are tabulated in Table 1. All experiments were done on a 3.0GHz Pentium IV computer with 1G of memory. Our tool parsed and analyzed around 4.8 million lines of code in 63,850 functions in under 20 hours. Function side-effect computation is not currently implemented in the version of the checker reported here. Loops are unrolled a maximum of two iterations based on the belief that most double lock errors manifest themselves by the second iteration. We have implemented an optimization that skips functions that have no lock primitives and do not call any other functions with non-trivial lock summaries. These functions are automatically given the trivial "No-Op" summary. We analyzed the remaining 23,927 lock related functions, and stored their summaries in a GDBM database.

We set the memory limit for each function to 700MB to prevent thrashing, and the CPU time limit to 90 seconds. Our tool failed to analyze 27 functions – some of which were written in assembly, and the rest due to internal failures of the tool. It failed to terminate on 442 functions in the kernel, largely due to resource constraints, with a small number of them due to implementation bugs in our tool. In every case we have investigated, resource exhaustion is caused by exceeding the capacity of an internal cache in Saturn. This represents a failure rate of $< 2\%$ on the lock-related functions.

Type	Bugs	FP	Warnings	Accuracy (Bug/Warning)
A	134	99	233	57%
B	45	22	67	67%
Total	179	121	300	60%

Table 2: Bug count.

Type	A	B	Total
Interprocedural	108	27	135
Intraprocedural	26	18	44
total	134	45	179

Table 3: Bug breakdown.

The result of the analysis consists of a bug report of 179 previously unknown errors[6] and a lock summary database for the entire kernel, which we describe in the subsections below.

6.1 Errors and False Positives

As described in Section 5.5, we detect two types of locking errors in Linux: double lock/unlock (Type A) and ambiguous output states (Type B). We tabulate the bug counts in Table 2.

The bugs and false positives are classified by manually inspecting the error reports generated by the tool. One caveat of this approach is that errors we diagnose may not be actual errors. To counter this, we only flag ones we are reasonably sure about. We have several years of experience examining Linux bugs, so the number of misdiagnosed errors is expected to be low.

Table 3 further breaks down the 179 bugs into intraprocedural versus interprocedural errors. We observe that more than three quarters of diagnosed errors are caused by misunderstanding of function interface constraints.

Table 4 classifies the false positives into six categories. The biggest category of false positives is caused by inadequate choice of predicates P_{in} and P_{out}. In a small number of widely called utility functions, input and output lock states are correlated with values passed in/out through the parameter, instead of the return value. To improve this, we need to detect the relevant predicates either by manual specification or by using a predicate abstraction algorithm similar to that used in SLAM or BLAST, which we will leave to future work. Another large source of false positives is the idiom that uses trylock operations as a way of querying the current state of the lock. This idiom is commonly used in assertions to make sure that a lock is held at a certain point. We believe a better way to accomplish this task is to use the lock query functions, which we model precisely in our tool. Fortunately, this usage pattern only occurs in a few macros, and can be easily identified during inspec-

[6]The bug reports are available online at `http://glide.stanford.edu/saturn/results/err{1,2}.php`

	Type A	Type B	Total
Predicates	26	16	42
Lock Assertions	21	4	25
Semaphores	22	0	22
Saturn Lim.	18	1	19
Readlocks	7	0	7
Others	5	1	7
Total	99	22	121

Table 4: False positives breakdown.

```
1 static void sscape_coproc_close(void *dev_info, int sub_device)
2 {
3         spin_lock_irqsave(&devc->lock,flags);
4         if (devc->dma_allocated) {
5                 sscape_write(devc, GA_DMAA_REG, 0x20);
6                 ...
7         ...
8 }
```

Figure 11: An interprocedural Type A error found in `sound/oss/sscape.c`.

```
1  int i2o_claim_device(struct i2o_device *d,
2                       struct i2o_handler *h)
3  {
4          down(&i2o_configuration_lock);
5          if (d->owner) {
6                  ...
7                  up(&i2o_configuration_lock);
8                  return -EBUSY;
9          }
10         ...
11         if(...) {
12                 ...
13                 return -EBUSY;
14         }
15         up(&i2o_configuration_lock);
16         return 0;
17 }
```

Figure 12: An intraprocedural Type B error found in `drivers/message/i2o/i2o_core.c`.

tion. The third largest source of false positives is counting semaphores. Depending on the context, semaphores can be used in Linux either as locks (with down being lock and up being unlock) or counters. Our tool treats all semaphores as locks, and therefore may misflag consecutive down/up operations as double lock/unlock errors. The remaining false positives are due to readlocks (where double locks are OK), and Saturn limitations in modeling arrays, void *, etc.

Figure 11 shows a sample interprocedural Type A error found by Saturn, where `sscape_coproc_close` calls `sscape_write` with &devc→lock held. However, the first thing `sscape_write` does is to acquire that lock again, resulting in a deadlock on multiprocessor systems.

Figure 12 gives a sample intraprocedural Type B error. There are two places where the function exits with return value -EBUSY: one with the lock held, and the other unheld. The programmer has forgotten to release the lock before returning at line 13.

We have filed the bug reports to the Linux Kernel Mailing List (LKML) and received confirmations and patches for a number of reported errors. To the best of our knowledge, Saturn is by far the most effective bug detection tool for Linux locking errors.

6.2 The Lock Summary Database

Synchronization errors are known to be difficult to reproduce and debug dynamically. To help developers diagnose reported errors, and also better understand the often subtle locking behavior in the kernel (e.g., lock states under error conditions), we have built a web interface for the Linux lock summary database[7] generated during the analysis.

Our own experience with the summary database has been pleasant. During inspection, we use the summary database extensively to match up the derived summary with the implementation code to determine whether a bug report is a false positive. In our experience the generated summaries accurately model the locking behavior of the function being analyzed. In fact, shortly after we filed these bugs, we logged more than a thousand queries to the summary database from the Linux community.

The summary database also reveals interesting facts about the Linux kernel. To our surprise, locking behavior is far from simple in Linux. More than 23,000 of the ∼63,000 functions in Linux directly or indirectly operate on locks. In addition, 8873 functions access more than one lock. There are 193 lock wrappers, 375 unlock wrappers, and 36 functions where the output state correlates with the return value. Furthermore, more than 17,000 functions directly or indirectly require locks to be in a particular state on entry.

We believe Saturn is the first automatic tool that successfully understands and documents any aspect of locking behavior in code the size of Linux.

7. RELATED AND FUTURE WORK

In this section, we discuss the relationship of Saturn to several other systems for bug finding and verification.

Saturn was inspired by the first author's previous work on *Meta Compilation* (MC) [10, 13] and our project is philosophically aligned with MC in that it is a bug detection, rather than a verification, system. In fact, Saturn began as an attempt to improve the accuracy of MC's flow sensitive but path insensitive analysis.

Under the hood, MC attaches finite state machines (FSM) to syntactic program objects (e.g., variables, memory locations, etc.) and uses an interprocedural data flow analysis to compute the reachability of the error state. Because conservative pointer analysis is often a source of false positives for bug finding purposes [12], MC simply chooses not to model pointers or the heap, thereby preventing false positives from spurious alias relationships by fiat. MC checkers use heuristics (e.g., separate FSM transitions for the true and false branches of "interesting" `if` statements) and statistical methods to infer some of the lost information. These techniques usually dramatically reduce false positive rates after several rounds of trial and error. However, they cannot fully compensate for the information lost during the analysis. For example, in the code below,

[7]The web interface is available online at `http://glide.stanford.edu/saturn/results/browse.php`

```
/* 1: data correlation */
if (x) spin_lock(&lock);
if (x) spin_unlock(&lock);
/* 2: aliasing */
l = &p->lock;
spin_lock(&p->lock);
spin_lock(l);
```

MC emits a spurious warning in the first case, and misses the error in the second. The first scenario occurs frequently in Linux, and an interprocedural version of the second is also prevalent.

Saturn can be viewed as both a generalization and simplification of MC because it uniformly relies on boolean satisfiability to model all aspects without special cases. The lock checker presented in Section 5.5 naturally tracks locks that are buried in the heap, or conditionally manipulated based on the values of certain predicates. In designing this checker, we focused on two kinds of Linux mutex errors that exhibited high rates of false positives in MC: double locking and double unlocking (2 errors and 23 false positives [10]). Our experiments show that Saturn's improved accuracy and summary-based interprocedural analysis allow it to better capture locking behavior in the Linux kernel and thus find more errors at a lower false positive rate.

While Blast, SLAM, and other software model checking projects have made dramatic progress and now handle hundreds of thousands of lines of code [4, 15, 14], these are whole-program analyses. ESP, a lower-complexity approach based on context-free reachability, is similarly whole-program [9]. In contrast, Saturn computes summaries function-by-function and, based on our experiments, scales to millions of lines of code and should in fact be able to scale arbitrarily, at least for checking properties that lend themselves to concise function summaries. In addition, Saturn has the precision of path-sensitive bit-level analysis within function bodies, which makes handling normally difficult-to-model constructs, such as type casts, easy. In fact, Saturn's code size is only about 25% of the comparable part of Blast (the most advanced software model checker available to us), which supports our impression that a SAT-based checker is easier to engineer. One weakness of Saturn with respect to these systems is its current handling of loops and recursion (which is the primary source of unsoundness). This is an area we intend to examine in future work.

CQual is a quite different, type-based approach to program checking [12, 2]. CQual's primary limitation is that it is path insensitive. In the locking application path sensitivity is not particularly important for most locks, but we have found that it is essential for uncovering the numerous `trylock` errors in Linux. CQual's strength is in sophisticated global alias analysis that allows for sound reasoning and relatively few false positives due to spurious aliases; another possible direction for future work is to integrate this alias analysis into Saturn.

PREfix [6] is a symbolic execution based static error-detection tool. It selectively simulates a set number of paths through a function using a solver that specializes in detecting memory errors such as NULL pointer dereferences and leaks. Information gathered during the simulation is summarized into interface constraints for the function, which are subsequently used during the analysis of its callers. Saturn employs a similar architecture in its summary-based bottom-up interprocedural analysis.

CBMC [19, 8] is a SAT-based bounded model checker for C. It has been used to verify hardware design specifications [19] and also a component of an air traffic control system [8]. Like Saturn, it translates C programs into boolean formulas by unrolling loops up to a given bound and uses a SAT solver to analyze relevant properties. However, there are a few key differences between these two systems. On a high level, CBMC is optimized as a whole-program assertion checker aimed at verifying C programs with hundreds of lines of code. Saturn is designed for compositional analysis of large systems with millions of lines of code. The difference in goals results in low-level technical differences in modeling C features. For example, CBMC translates assignments as assumptions and feeds them to the SAT solver regardless of whether or not they are relevant to the properties being checked. This approach is not usually a problem because specification code is by design an abstraction of the implementation and thus should likely contain relevant information. The same cannot be said about the unmodified Linux kernel. In Linux, most of the code analyzed (e.g., complex logic in handling logs in the file system) is irrelevant to the property (e.g., locking behavior) of interest. Saturn models assignments with a map recording the binding of variables to their boolean representations, and Saturn only commits the boolean constraints to the SAT solver if they are relevant to the query. This design dramatically cuts the time spent in the solver. Saturn's modeling of pointers using guarded location sets also lends itself well to the automatic construction of the checking environment.

Closer to Saturn is Magic [7], a compositional checking tool that verifies a form of conformance relationship called weak simulation between two labeled transition systems (LTS): one abstracted from the program by a predicate abstraction tool like SLAM or BLAST, and the other written by the programmer as a form of specification. Checking is done by reducing the conformance query to the satisfiability of a special form of boolean formulas (weakly negated HORN formulas) that can be solved in linear time. Magic also focuses on checking finite state machine properties. The basic difference between Saturn and Magic is that Magic is conceived of as a tool to check user-supplied specification, while Saturn is an inference system.

8. CONCLUSIONS

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 have demonstrated the utility of tool with a lock checker for Linux, finding in the process 179 unique locking errors in the Linux kernel.

Acknowledgments

We thank Andy Chou for thoughtful discussions and significant contributions to an earlier SAT-based analysis effort. We also thank Ted Kremenek, Mayur Naik, Paul Twohey, Junfeng Yang and the anonymous reviewers for providing valuable comments on previous drafts of this paper.

9. 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 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 129–140, June 2003.

[3] T. Ball, B. Cook, V. Levin, and S. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *Proceedings of Fourth International Conference on Integrated Formal Methods*. Springer, 2004.

[4] 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.

[5] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug. 1986.

[6] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice & Experience*, 30(7):775–802, June 2000.

[7] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 385–395, 2003.

[8] 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.

[9] 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.

[10] 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.

[11] C. Flanagan and S. Freund. Type inference against races. In *Proceedings of 11th Static Analysis Symposium*, Verona, Italy, Aug. 2004.

[12] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, June 2002.

[13] 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.

[14] T. A. Henzinger, R. Jhala, and R. Majumdar. Lazy abstraction. In *Proceedings of the 29th Annual Symposium on Principles of Programming Languages*, January 2002.

[15] 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.

[16] D. Jackson. Automating first-order relational logic. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2000.

[17] 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.

[18] S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2003.

[19] D. Kroening, E. Clarke, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of DAC 2003*, pages 368–371. ACM Press, 2003.

[20] X. Leroy, D. Doligez, J. Garrigue, and J. Vouillon. The Objective Caml system. Software and documentation available on the web, `http://caml.inria.fr`.

[21] J. Lind-Nielsen. BuDDy, a binary decision diagram package. `http://www.itu.dk/research/buddy/`.

[22] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 39th Conference on Design Automation Conference*, June 2001.

[23] 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.

[24] Y. Xie and A. Chou. Path sensitive analysis using boolean satisfiability. Technical report, Stanford University, Nov. 2002.

[25] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of International Conference on Computer-Aided Design*, San Jose, CA, Nov. 2001.