Saturn: A Scalable Framework for Error Detection using Boolean Satisfiability

Yichen Xie and Alex Aiken
Stanford University

This article presents Saturn, a general framework for building precise and scalable static error detection systems. Saturn exploits recent advances in boolean satisfiability (SAT) solvers and 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 dataflow 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 have experimentally validated our approach by conducting two case studies involving a Linux lock checker and a memory leak checker. Results from the experiments show that our system scales well, parallelizes well, and finds more errors with fewer false positives than previous static error detection systems.

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.

Additional Key Words and Phrases: Program analysis, error detection, boolean satisfiability.

1. INTRODUCTION

This article presents Saturn[1], a software error-detection framework based on exploiting recent advances in solving boolean satisfiability (SAT) constraints.

At a high level, Saturn works by transforming commonly used program constructs into boolean constraints and then using a SAT solver to infer and check

[1] SATisfiability-based failURe aNalysis.

This research is supported by National Science Foundation grant CCF-1234567.
This article combines techniques and algorithms presented in two previous conference papers by the authors, published respectively in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005)* and *Proceedings of the 5th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*.
Authors' Address: Yichen Xie and Alex Aiken, Computer Science Department, Stanford University, Stanford, CA 94305; E-mail: {yxie,aiken}@cs.stanford.edu.

program properties. Compared to previous error detection tools based on data flow analysis or abstract interpretation, our approach has the following advantages:

(1) *Precision:* Saturn's modeling of loop-free code is faithful down to the bit level, and is therefore considerably more precise than most abstraction-based approaches where immediate information loss occurs at abstraction time. In the context of error detection, the extra precision translates into added analysis power with less confusion, which we demonstrate by finding many more errors with significantly fewer false positives than previous approaches.

(2) *Flexibility:* Traditional techniques rely on a combination of carefully chosen abstractions to focus on a class of properties effectively. Saturn, by exploiting the expressive power of boolean constraints, uniformly models many language features and can therefore serve as a general framework for a wider range of analyses. We demonstrate the flexibility of our approach by encoding two property checkers in Saturn that traditionally require distinct sets of techniques.

However, SAT-solving is NP-complete, and therefore incurs a worst-case exponential time cost. Since Saturn aims at checking large programs with millions of lines of code, we employ two techniques to make our approach scale. Intraprocedurally, our encoding of program constructs as boolean formulas is substantially more compact than previous approaches (Section 2). While we model each bit path sensitively as in [Xie and Chou 2002; Kroening et al. 2003; Clarke et al. 2004], several techniques achieve a substantial reduction in the size of the SAT formulas Saturn must solve (Section 3).

Interprocedurally, Saturn computes a concise summary, similar to a type signature, for each analyzed function. The summary-based approach 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, lending itself well to checking complex program behaviors (see Section 5.2 for an example).

Summary-based interprocedural analysis also enables parallelization. Saturn processes each function separately and the analysis can be carried out in parallel, subject only to the ordering dependencies of the function call graph. In Section 6.8, we describe a simple distributed architecture that harnesses the processing power of a heterogeneous cluster of roughly 80 unloaded CPUs. Our implementation dramatically reduces the running time of the leak checker on the Linux kernel (5MLOC) from over 23 hours to 50 minutes.

We present experimental results to validate our approach (Sections 5 and 6). Section 5 describes the encoding of temporal safety properties in Saturn and presents an interprocedural analysis that automatically infers and checks such properties. 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). Section 6 gives a context- and path-sensitive escape analysis of dynamically allocated objects. Both checkers find more errors than previous approaches with significantly fewer false positives.

One thing that Saturn is not, at least in its current form, is a verification framework. Tools such as CQual [Foster et al. 2002] 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). In this paper, Saturn is used as a bug finding framework in the spirit of MC [Hallem et al. 2002], 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.

The rest of the article is organized as follows: Section 2 presents the Saturn language and its encoding into boolean constraints. Section 3 discusses a number of key improvements to the encoding that enable efficient checking of open programs. Section 4 gives a brief outline of how we use the Saturn framework to build modular checkers for software. Sections 5 and 6 are two case studies where we present the details of the design and implementation of two property checkers. We describe sources of unsoundness for both checkers in Section 7. Related works is discussed in Section 8 and we conclude with Section 9.

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 integers, structures, pointers, and handles the arbitrary control flow[2] found in C. We begin with a language and encoding that handles only integer program values (Section 2.1) and gradually add features until we have presented the entire framework: intraprocedural control flow including loops (Section 2.2), structures (Section 2.3), pointers (Section 2.4), and finally attributes (Section 2.5). In Section 3 we consider some techniques that substantially improve the performance of our encoding.

2.1 Modeling Integers

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

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.

[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 [Aho et al. 1986].

$\boxed{\textsf{Language}}$

$$
\begin{array}{rl}
\textsf{Type } (\tau) & ::= (n, \textsf{signed} \mid \textsf{unsigned}) \\
\textsf{Obj } (o) & ::= v \\
\textsf{Expr } (e) & ::= \textsf{unknown}(\tau) \mid \textsf{const}(n, \tau) \mid o \mid \textsf{unop } e \mid e_1 \textsf{ binop } e_2 \mid (\tau)\ e \mid \textsf{lift}_e(c, \tau) \\
\textsf{Cond } (c) & ::= \textsf{false} \mid \textsf{true} \mid \neg\ c \mid e_1 \textsf{ comp } e_2 \mid c_1 \wedge c_2 \mid c_1 \vee c_2 \mid \textsf{lift}_c(e) \\
\textsf{Stmt } (s) & ::= o \leftarrow e \mid \textsf{assert}(c) \mid \textsf{assume}(c) \mid \textsf{skip}
\end{array}
$$

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

$\boxed{\textsf{Representation}}$

$$
\begin{array}{rl}
\textsf{Rep } (\beta) & ::= [b_{n-1} \dots b_0]_s \quad \text{where } s \in \{\textsf{signed}, \textsf{unsigned}\} \\
\textsf{Bit } (b) & ::= 0 \mid 1 \mid x \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid \neg b
\end{array}
$$

Fig. 1. Modeling integers in SATURN.

$\boxed{\textsf{Expressions}}$

$$\dfrac{\beta = \psi(v)}{\psi \vdash v \overset{\underline{\mathtt{E}}}{\Rightarrow} \beta} \quad \textsf{scalar}$$

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

$$\dfrac{\begin{array}{c}\psi \vdash e \overset{\underline{\mathtt{E}}}{\Rightarrow} [b_{n-1} \dots b_0]_x \quad \tau = (m, s) \\ b'_i = \begin{cases} b_i & \text{if } 0 \leq i < n \\ 0 & \text{if } s = \textsf{unsigned} \text{ and } n \leq i < m \\ b_{n-1} & \text{if } s = \textsf{signed} \quad \text{and } n \leq i < m \end{cases}\end{array}}{\psi \vdash (\tau)\ e \overset{\underline{\mathtt{E}}}{\Rightarrow} [b'_{m-1} \cdots b'_0]_s} \quad \textsf{cast}$$

$$\dfrac{(n, s) = \tau \quad \psi \vdash c \overset{\underline{\mathtt{C}}}{\Rightarrow} b}{\psi \vdash \textsf{lift}_e(c, \tau) \overset{\underline{\mathtt{E}}}{\Rightarrow} [\underbrace{00 \cdots 0}_{n-1} b]_s} \quad \textsf{lift}_e$$

$$\dfrac{\begin{array}{c}\psi \vdash e \overset{\underline{\mathtt{E}}}{\Rightarrow} [b_{n-1} \dots b_0]_s \\ \psi \vdash e' \overset{\underline{\mathtt{E}}}{\Rightarrow} [b'_{n-1} \cdots b'_0]_s\end{array}}{\psi \vdash e \texttt{ band } e' \overset{\underline{\mathtt{E}}}{\Rightarrow} [b_{n-1} \wedge b'_{n-1} \cdots b_0 \wedge b'_0]_s} \quad \textsf{and}$$

$\boxed{\textsf{Conditionals}}$

$$\dfrac{\psi \vdash e \overset{\underline{\mathtt{E}}}{\Rightarrow} [b_{n-1} \dots b_0]_s}{\psi \vdash \textsf{lift}_c(e) \overset{\underline{\mathtt{C}}}{\Rightarrow} \bigvee_i b_i} \quad \textsf{lift}_c$$

$\boxed{\textsf{Statements}}$

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

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

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

Fig. 2. The translation of integers.

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 encoding. 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 integer variables.

The encoding for a representative selection of constructs is shown in Figure 2; omitted cases introduce no new ideas. The rules for expressions 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 encoded as a vector of boolean expressions β.

The encoding 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:

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

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. Without going into details, we explain the conceptual meaning of a guard using the following example:

```
if (c) {s1;s2} else s3;
s4;
```

Statements s_1 and s_2 are executed if c is true, so the guard for both statements is the boolean encoding of c. Similarly, s_3's guard is the encoding of $\neg c$. Statement s_4 is reached from both branches of the `if` statement and therefore its guard is the disjunction of the guards from the two branches: $(c \vee \neg c) = \mathtt{true}$.

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 $\mathcal{G}$ is the guard of the `assert` and b is the encoding of the condition c.

The overall effect of the encoding is to perform 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 encoding is straightforward interpretation 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 encoding may be an arbitrary boolean expression over such variables.

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

Fig. 3. Merging control-flow paths.

2.2 Control Flow

We represent function bodies as control-flow graphs, which we define informally. For the purpose of this section, we assume loop-free programs. Loops are handled in a variety of ways which are described at the end of this section. Each statement s is a node in the control-flow 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 encoding 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 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 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.

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

$\boxed{\text{Language}}$

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

$\boxed{\text{Shorthand}}$

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

$\boxed{\text{Representation}}$

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

$\boxed{\text{Translation}}$

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

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

$$\frac{\begin{array}{c} \psi \vdash e \overset{\text{E}}{\Rightarrow} \beta \\ \psi' = \text{RecAssign}(\psi, o, \beta) \end{array}}{\mathcal{G}, \psi \vdash (o \leftarrow e) \overset{\text{S}}{\Rightarrow} \langle \mathcal{G}; \psi' \rangle} \ \text{assign-struct}$$

Fig. 4. The translation of structures.

As mentioned in Section 1, the two checkers described in this paper treat loops unsoundly. One technique we adopt is to simply unroll a loop a fixed number of times and remove backedges from the control-flow graph. Thus, every function body is represented by an acyclic control-flow graph. Another transformation is called *havoc'ing*, which we discuss in detail in the context of the memory leak checker (Section 6).

While our handling of loops is unsound, we have found it to be useful in practice (see Section 5.6 and 6.9).

2.3 Structures

The program syntax and the encoding 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 encoding is pointers. The discussion is divided into three parts: in Section 2.4.1, we introduce a concept called *Guarded Location Set* (GLS) to capture path-sensitive points-to information. We extend the representation with type casts and polymorphic locations in Section 2.4.2 and discuss the rules in detail in Section 2.4.3.

2.4.1 *Guarded Location Sets.* Pointers in SATURN are modeled with *Guarded Location Sets* (GLS). A GLS represents the set of locations a pointer could refer-

$\boxed{\text{Language}}$

$$\begin{array}{rrl}
\text{Type} & (\tau) & ::= \tau* \mid \textsf{void*} \mid \ldots \\
\text{Obj} & (o) & ::= p \mid \ldots \\
\text{Deref} & (m) & ::= (*p).f_1.\cdots.f_n \qquad (n \geq 0) \\
\text{Expr} & (e) & ::= \textsf{null} \mid \&o \mid \&m \mid \ldots \\
\text{Stmt} & (s) & ::= \textsf{load}(m, o) \mid \textsf{store}(m, e) \mid \textsf{newloc}(p) \mid \ldots
\end{array}$$

$\boxed{\text{Address}}$

Addr (σ) $::= \hat{1} \mid \hat{2} \mid \ldots$
AddrOf : Obj $\mapsto$ Addr
(Constraint: no two objects of the same type share the same address)

$\boxed{\text{Representation}}$

$$\begin{array}{rrl}
\text{Loc} & (l) & ::= \textsf{null} \mid o \mid \sigma \\
\text{Rep} & (\beta) & ::= \{\!|\ (\mathcal{G}_0, l_0), \ldots, (\mathcal{G}_k, l_k)\ |\!\} \mid \ldots
\end{array}$$

$\boxed{\text{Translation}}$

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

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

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

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

$$\frac{\begin{array}{c} l = \begin{cases} o & \text{if } p \text{ is of type } \tau* \\ \sigma & \text{if } p \text{ is of type } \textsf{void*} \end{cases} \\ \beta = \{\!|\ (\textsf{true}, l)\ |\!\} \text{ and } o \text{ or } \sigma \text{ fresh} \end{array}}{\mathcal{G}, \psi \vdash \textsf{newloc}(p) \overset{\mathtt{S}}{\Rightarrow} \langle \mathcal{G}; \psi[p \mapsto \beta] \rangle} \quad \textsf{newloc}$$

$$\frac{\begin{array}{c} \psi(p) = \{\!|\ (\mathcal{G}_0, \textsf{null}), \overline{(\mathcal{G}_i, \sigma_i)}\ |\!\} \\ \text{type of } o_i = \tau \text{ and } \textsf{AddrOf}(o_i) = \sigma_i \end{array}}{\psi \vdash (\tau*)p \overset{\mathtt{E}}{\Rightarrow} \{\!|\ (\mathcal{G}_0, \textsf{null}), \overline{(\mathcal{G}_i, o_i)}\ |\!\}} \quad \textsf{cast-from-void*}$$

$$\frac{\begin{array}{c} \psi(p) = \{\!|\ (\mathcal{G}_0, \textsf{null}), \overline{(\mathcal{G}_i, o_i)}\ |\!\} \\ \textsf{AddrOf}(o_i) = \sigma_i \end{array}}{\psi \vdash (\textsf{void*})p \overset{\mathtt{E}}{\Rightarrow} \{\!|\ (\mathcal{G}_0, \textsf{null}), \overline{(\mathcal{G}_i, \sigma_i)}\ |\!\}} \quad \textsf{cast-to-void*}$$

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

Fig. 5. Pointers and guarded location sets.

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

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

$$\textsf{MergeEnv}\left(\overline{(\mathcal{G}_i, \psi_i)}\right) = \langle \bigvee_i \mathcal{G}_i; \psi \rangle \qquad \text{where} \begin{cases} \psi(v) = \textsf{MergeScalar}\left(v, \overline{(\mathcal{G}_i, \psi_i)}\right) \\ \psi(p) = \textsf{MergePointer}\left(p, \overline{(\mathcal{G}_i, \psi_i)}\right) \end{cases}$$

Fig. 6. Control-flow merges with pointers.

ence at a particular program point. To maintain path-sensitivity, a boolean guard is associated with each location in the GLS and represents the condition under which the points-to relationship holds. We write a GLS as $\{\!|\ (\mathcal{G}_0, l_0), \ldots, (\mathcal{G}_n, l_n)\ |\!\}$. Special braces ($\{\!|\ \ |\!\}$) distinguish GLSs from other sets. We illustrate GLS with an example. but delay technical discussion until Section 2.4.3.

```
1  if (c) p = &x;      /* p : {| (true, x) |} */
2  else p = &y;        /* p : {| (true, y) |} */
3  *p = 3;         /* p : {| (c, x), (¬c, y) |} */
```

In the true branch, the GLS for p is $\{\!|\ (\mathsf{true}, x)\ |\!\}$, meaning p always points to x. Similarly, $\psi(p)$ evaluates to $\{\!|\ (\mathsf{true}, y)\ |\!\}$ in the false branch. At the merge point, branch guards are added to the respective GLSs and the representation for p becomes $\{\!|\ (c, x), (\neg c, y)\ |\!\}$. Finally, the store at line 3 makes a parallel assignment to x and y under their respective guards (i.e., if (c) x = 3; else y = 3;).

To simplify technical discussion, we assume locations in a GLS occur at most once—redundant entries $(\mathcal{G}, l)$ and $(\mathcal{G}', l)$ are merged into $(\mathcal{G} \vee \mathcal{G}', l)$. Also, we assume the first location l_0 is always null (we use the false guard for $\mathcal{G}_0$ if necessary).

2.4.2 *Polymorphic Locations and Type Casts.* The GLS representation models pointers to concrete objects with a single known type. However, it is common for heap objects to go through multiple types in C. For example, in the following code,

```
1  void *malloc(int size);
2  p = (int *)malloc(len);
3  q = (char *)p;
4  return q;
```

the memory block allocated at line 2 goes through three different types. These types all have different representations (i.e., different numbers of bits) and thus need to be modeled separately, but the analysis must understand that they refer to the same location. We need to model: 1) the polymorphic pointer type void*, and 2) cast operations to and from void*. Casts between incompatible pointer types (e.g. from int* to char*) can then be modeled via an intermediate cast to void*.

We solve this problem by introducing *addresses* (Addr), which are symbolic identifiers associated with each unique memory location. We use a mapping AddrOf : Obj → Addr to record the addresses of objects. Objects of different types share the same address if they start at the same memory location. In the example above, p and q point to different objects, say o_1 of type int and o_2 of type char, and o_1 and o_2 must share the same address (i.e. $\mathsf{AddrOf}(o_1) = \mathsf{AddrOf}(o_2)$). Furthermore, an address may have no associated concrete objects if it is referenced only by a pointer of type void* and never dereferenced at any other types. In other words, the inverse mapping AddrOf^{-1} may not be defined for some addresses. Using guarded location sets and addresses, we can now describe the encoding of pointers in detail.

2.4.3 *Encoding Rules.* Figure 5 defines the language and encoding rules for pointers. Locations in the GLS can be 1) null, 2) a concrete object o, or 3) an address σ of a polymorphic pointer (void*). We maintain a global mapping AddrOf from objects to their addresses and use it in the cast rules to convert pointers to and from void*.

The rules work as follows. Taking the address of an object (get-addr-{obj,mem}) constructs a GLS with a single entry–the object itself with guard true. The newloc rule creates a fresh object or address depending on the type of the target pointer and binds the GLS containing that location to the target pointer in the environment ψ. Notice that Saturn does not have a primitive modeling explicit deallocation. Type casts to void* lift entries in the GLS to their addresses using the AddrOf mapping, and casts from void* find the concrete object of the appropriate type in the AddrOf mapping to replace addresses in the GLS. Finally, the store rule models indirect assignment through a pointer, possibly involving field dereferences, by combining the results for each possible location the pointer could point to. The pointer is assumed to be non-null by adding $\neg\mathcal{G}_0$ to the current guard (recall $\mathcal{G}_0$ is the guard of null in every GLS). Notice that the store rule requires concrete locations in the GLS as one cannot assign through a pointer of type void*. Loading from a pointer is similar.

2.5 Attributes

Another feature in Saturn is *attributes*, which are simply annotations associated with non-null Saturn locations (i.e. structs, integer variables, pointers, and addresses). We use the syntax o#attrname to denote the attrname attribute of object o.

The definition and encoding of attributes is similar to struct fields except that it does not require predeclaration, and attributes can be added during the analysis as needed. Similar to struct fields, attributes can also be accessed indirectly through pointers.

We omit the formal definition and encoding rules because of their similarity to field accesses. Instead, we use an example to illustrate attribute usage in analysis.

```
1  (*p)#escaped <- true;
2  q <- (void *) p;
3  assert ((*q)#escaped == (*p)#escaped);
```

In the example above, we use the store statement at line 1 to model the fact that the location pointed to by p has escaped. The advantage of using attributes here is that they are attached to addresses and preserved through pointer casts—thus the assertion at line 3 holds.

3. DISCUSSION AND IMPROVEMENTS

In this section, we discuss how our encoding reduces the size of satisfiability queries by achieving a form of program slicing (Section 3.1). 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 of concern. Slicing is commonly done by computing control and data dependencies and preserving only the statements that the property depends on. We show that our encoding 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 encoding rules in Section 2, we see that the encoding 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. No extra constraints are included. In large programs, properties of interest often depend on a small portion of the code analyzed, therefore 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 [Ball and Rajamani 2001] and Blast [Henzinger et al. 2003] 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) [Ball et al. 2004]. 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 [Khurshid et al. 2003] and, conceptually, to lazy evaluation in languages such as Haskell. 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 $\mathsf{DefVal}(v)$ for all v. Operationally, DefVal is applied on demand, when uninitialized objects are first accessed 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 approximation of the return values and side-effects of function calls (Section 5.3).

In our implementation, we model integers 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 previously unseen pointer dereference.[4] 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

[4] In the implementation, $\mathsf{DefVal(p)}$ returns $\{\!|\ (\mathcal{G}, \mathsf{null}), (\neg\mathcal{G}, o)\ |\!\}$, where $\mathcal{G}$ is an unconstrained boolean 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.

performs an accurate path-sensitive intraprocedural analysis, including respecting alias relationships, on that pointer.

3.3 Using BDDs for Guards

Consider the following code fragment:

```
if (c) { ... } else { ... }
s;
```

After conversion to a control-flow graph, there are two paths reaching the statement `s` with guards `c` and ¬`c`. Thus the guard of `s` is `c`∨¬`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 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 [Bryant 1986]. We give each condition (which may be a complex expression) a name and use a BDD to represent the boolean combination of all condition names 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 ¬`c` is represented by `true`. In our encoding 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.

4. BUILDING MODULAR PROPERTY CHECKERS UNDER SATURN

The Saturn framework we have described so far can be applied directly to checking simple properties such as assertions. While other program behavior can be encoded and checked under the current scheme, there are two main limitations that prevent it from being applied to complex properties in large systems.

(1) *Function calls.* Saturn, like many other SAT-based techniques, does not directly model function calls. A common solution among SAT-based assertion checkers is inlining. However, although we employ a number of optimizations in our transformation such as slicing, the exponential time cost of SAT-solving means that inlining will not be practical for large software systems.

(2) *Execution environment.* Assertion checking commonly requires a closed program. However, many software systems are open programs whose environment is a complex combination of user input and component interdependencies. Modeling the environment for such programs often requires extensive manual effort that is both costly and error prone.

Our solution is based on SATURN's ability to not only *check* program properties, but also *infer* them by formulating SAT queries that can be solved efficiently. The latter ability solves the two problems mentioned above.

First, inference enables modular analyses[5] that scale. With appropriate abstractions, the checker can summarize a function's behavior with respect to a property into a concise summary. This summary, in turn, can be used in lieu of the full function body at the function's call sites, which prevents the exponential growth in the cost of analysis.

Secondly, by making general enough assumptions about the execution environment, summaries capture the behavior of a function under all (or, for error detection purposes, a common subset of) runtime scenarios. This alleviates the requirement of having to close the environment.

An added benefit of the modular approach is that it enables local reasoning during error inspection. Instead of following long error traces which may involve multiple function calls, human-readable function summaries give information about the assumptions made for each of the callees in the current function. Therefore, the user can focus on one function at a time when inspecting error reports. In our experience, we find it much easier to confirm errors and identify false positives with the help of function summaries.

Based on the modular approach, we briefly outline a four step process by which we construct property checkers under SATURN:

(1) First of all, we model the property we intend to check with program constructs in SATURN. For example, finite state machines (FSM) can be encoded by attaching integer state fields to program objects to track their current states. State transitions are subsequently modeled with conditional assignments, and checking is done by ensuring that the error state is not reached at the end of the program—a task easily accomplished with SAT queries on the final program state.

(2) The next step is to design the function summary representation. A good summary is one that is both concise for scalability and expressive enough to adequately describe the relevant properties of function behavior. Striking the right balance often takes several iterations of design. For example, in designing the FSM checking framework, we started with a simple summary that records the set of feasible state transitions across the function, but found it to be inadequate for Linux lock checking because of interprocedural data dependencies. We observed that the output lock state often correlates with the return value of the function and remedied the situation by simply including the return value in our summary design.

(3) The third step is to design an algorithm that infers and applies function summaries. As mentioned above, inference is done by automatically inserting SAT queries at appropriate program points. For example, we can infer the set of possible state transitions by querying, at the end of each function, the satis-

[5] Here, modular analysis is defined in two senses: 1) the ability to infer and check open program modules independent of their usage; and 2) the ability to summarize results of analyzed modules so as to avoid redundant analysis.

fiability of all possible combinations of input and output states. The feasible (i.e., satisfiable) subset is included in the function summary.[6]

(4) Finally, we run the checker on a number of real-world applications, and inspect the analysis results. During early design iterations, the results often point to inaccuracies in the property encoding (Step 1), inadequacies in the summary design (Step 2), or inefficiencies in the inference algorithm (Step 3). We use that as feedback to improve the checker in the next iteration.

Following the four step process, we designed and implemented two property checkers for large open source software: a Linux lock checker and a memory leak checker. We present the details of the construction and experiments in the following two sections.

5. CASE STUDY I: 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 describe our implementation of an interprocedural lock checker (Section 5.5) and end with experimental results (Section 5.6).

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, parameters, and the function's result. Therefore, shared objects can be named using a path from one of these three 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

[6] This is a simplification of the actual summary inference algorithm, which takes into account function side-effects and return-value state-transition correlations. We describe the full algorithm in Section 5.2.

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

Fig. 7. Function summary representation.

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:

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

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

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

Note that the result of the conversion is in $\textsf{Obj}^{\textsf{ext}}$, which is defined as Obj (Section 2) extended with pointer dereferences. The extra dereference operations can be transformed away by introducing temporary variables and explicit load/store operations. We omit the details of this transformation for brevity.

The inverse conversion is more involved, since there may be multiple aliases of the same object in the program. We incrementally construct the $[\![\cdot]\!]^{-1}_{\textsf{args}}$ 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 $\textsf{param}_0$, 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 3.2). 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 $(*\textsf{param}_0).\textsf{lock}$.

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 propositions P_{in},

—a set of output propositions 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.

The checker need only supply the set of FSM states and the set of input and output propositions (i.e. $\mathcal{S}$, P_{in}, and P_{out}); 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 (P_{out}) propositions. 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.

—$\mathsf{incond} \in 2^{|P_{in}|}$ denotes the pre-condition of the FSM transition: $(\bigwedge_{i \in \mathsf{incond}} p_i) \wedge (\bigwedge_{i \notin \mathsf{incond}} \neg p_i)$ where $\{p_1, \ldots, p_n\} = P_{in}$. It specifies one of the 2^n possible valuations of the input propositions, and 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 similarly defined as 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, given appropriate input and output propositions (i.e., P_{in} and P_{out}), we are able to express (and automatically infer) its behavior using our summary representation (i.e., M and R). 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 encode the call into a set of statements simulating the observable effects of the function. The encoding, 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 (lines 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 propositions 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 use an assume statement to rule out impossible combinations of input and output propositions (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). The proposition incond denotes the condition $(\bigwedge_{i \in \mathsf{incond}} \widehat{p_i}) \wedge (\bigwedge_{i \notin \mathsf{incond}} \neg \widehat{p_i})$;

```
void complex_wrapper(spinlock_t *l, int flag, int *success)
{
    /* spin_trylock returns non-zero on successful
       acquisition of the lock; 0 otherwise. */
    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: $P_{in} = \{\}$ $\quad P_{out} = \{\}$
Output: $M = \{*\mathsf{param}_0\}$
$R = \{ \ (*\mathsf{param}_0, \_\,, \mathsf{Unlocked}, \_\,, \mathsf{Locked}),$
$\qquad (*\mathsf{param}_0, \_\,, \mathsf{Locked}, \_\,, \mathsf{Error})\}$

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

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

Fig. 8. Sample function summaries for the locking property.

the condition for outcond is symmetric. Notice that since incond and outcond are a valuation of all input and output propositions, no two transitions on the same state machine should be enabled simultaneously. Violations of this property can be attributed to either an inadequate choice of input and output propositions, or a bug in the program (e.g., Type B errors in the Linux lock checker–Section 5.5).

There is one aspect of the encoding that is left unspecified in the description, which is the unknown values used to model the side-effects of the function call. For integer values, we use the 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. The current implementation uses fresh locations for modified pointers.

Assumptions

$\Sigma(f) = \langle P_{in}, P_{out}, M, R\rangle$

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

Instrumentation

```
(* Stage 1: Preparation *)
(* save useful program states *)
$\widehat{p}_1 \leftarrow p_1; \ldots; \widehat{p}_n \leftarrow p_m;$
$\widehat{\mathsf{sm}}_1 \leftarrow \mathsf{sm}_1; \ldots; \widehat{\mathsf{sm}}_l \leftarrow \mathsf{sm}_l;$
(* account for the side-effects of $f$ *)
$o_1 \leftarrow \mathsf{unknown}(\tau_{o_1}); \ldots; o_k \leftarrow \mathsf{unknown}(\tau_{o_k});$
(* save the values of output propositions *)
$q'_1 \leftarrow q_1; \ldots; q'_n \leftarrow q_n;$
(* rule out infeasible comb. of incond and outcond *)
$\mathsf{assume}(\bigvee_i (\mathsf{sm}_i = s_i \wedge \mathsf{incond}_i \wedge \mathsf{outcond}_i));$

(* Stage 2: Transitions *)
(* record state transitions after the function call *)
if $(\widehat{\mathsf{sm}}_1 = s_1 \wedge \mathsf{incond}_1 \wedge \mathsf{outcond}_1)$ $\mathsf{sm}_1 \leftarrow s'_1;$
  ...
if $(\widehat{\mathsf{sm}}_l = s_l \wedge \mathsf{incond}_l \wedge \mathsf{outcond}_l)$ $\mathsf{sm}_l \leftarrow s'_l;$
```

Fig. 9. Summary application.

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

Fig. 10. Summary inference.

5.4 Summary Inference

This section 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's intermediate language:

(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 inference 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 inference algorithm. The input to the algorithm is a set of input (P_{in}) and output (P_{out}) propositions. The inference process 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 inference 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.

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

As the reader may notice, summary inference requires many SAT queries and can be computationally expensive when solved individually. Fortunately, these queries share a large set of common constraints encoding the function control and data flow. In fact, the only difference among the queries are constraints that describe the different combinations of input/output propositions and initial/final state pairs for each state machine. We exploit this fact by taking advantage of incremental solving capabilities in modern SAT solvers. Incremental solving algorithms share and reuse information learned (e.g., using conflict clauses) in the common parts of the queries and can considerably speed up SAT solving time for similar queries. In practice, we observe that 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 supervisor mode, and synchronization bugs can thus cause crashes or hangs that result in data loss 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 overcome 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 [Aiken et al. 2003]. MC [Hallem et al. 2002] 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 "harnesses" that construct the checking environment has proven to be a non-trivial task in traditional model checkers [Ball et al. 2004].
- **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, 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 5.6, 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: $\{\mathsf{Locked}, \mathsf{Unlocked}, \mathsf{Error}\}$. To accurately model trylocks, we define $P_{out} = \{\mathsf{lift}_c(\mathsf{ret\_val})\}$ for functions that return integers or pointers. Tracking this proposition 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. We define P_{out} 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 states with no observable difference (w.r.t. P_{out}, which is

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 I. Performance statistics on a single processor Pentium IV 3.0G desktop with 1GB memory.

$\mathsf{lift}_c(\mathsf{ret\_val})$) in the return value. These bugs are commonly caused by missed operations to restore lock states on error paths.[7]

5.6 Experimental Results

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 once in an arbitrary order, which might result in imprecision in inferred summaries. A more precise approach would require unwinding the recursion as we do for loops, until a fixed point is reached for function summaries in the SCC. However, our experiments indicate that recursion has little impact on the precision of inferred lock summaries, and therefore we adopt the simpler approach in our implementation.

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 I. All experiments were done on a single processor 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

[7] One can argue that Type B errors are rather a manifestation of the restricted sets of predicates used for the analysis; a more precise way of detecting these bugs is to allow ambiguous output states in the function summary, and report bugs in calling contexts where only one of the output states is legal. Practically, however, we find that this restriction is a desirable feature that allows us to exploit domain knowledge about lock usage in Linux, and thus helps the analysis to pinpoint more accurately the root cause of a bug.

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

Table II. Number of bugs found in each category.

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

Table III. Breakdown of intra- and inter-procedural bugs.

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. The tool also 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.

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

5.6.1 *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 II.

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 III 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 IV classifies the false positives into six categories. The biggest category of false positives is caused by inadequate choice of propositions 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

	Type A	Type B	Total
Propositions	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 IV. Breakdown of false positives.

```
static void sscape_coproc_close(void *dev_info, int sub_device)
{
        spin_lock_irqsave(&devc->lock,flags);
        if (devc->dma_allocated) {
                sscape_write(devc, GA_DMAA_REG, 0x20); // bug here
                ...
        ...
}

static void sscape_write(struct sscape_info *devc, int reg, int data)
{
        ...
        spin_lock_irqsave(&devc->lock,flags); // acquires the lock
}
```

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

value. To improve this situation, we need to detect the relevant propositions either by manual specification or by using a predicate abstraction algorithm similar to that used in SLAM or BLAST. Another large source of false positives is an 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 querying functions, which we model precisely in our tool. Fortunately, this usage pattern only occurs in a few macros, and can be easily identified during inspection. 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 resource 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 unmodeled features such as arrays.

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

```
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 }
```

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

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.

5.6.2 *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 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 confirm errors and identify false positives. 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.

6. CASE STUDY II: THE LEAK DETECTOR

In this section, we present a static memory leak detector based on the path sensitive pointer analysis in Saturn. We target one important class of leaks, namely neglecting to free a newly allocated memory block before all its references go out of scope. These bugs are commonly found in error handling paths, which are less likely to be covered during testing. This second study is interesting in its own right as an effective memory leak detector, and as evidence that Saturn can be used to analyze a variety properties.

The rest of the section is organized as follows: Section 6.1 gives examples illustrating the targeted class of bugs and the analysis techniques required. We briefly outline the detection algorithm in Section 6.2 and give details in Sections 6.3, 6.4, and 6.5. Handling the unsafe features of C is described in Section 6.7. Section 6.8 describes a parallel client/server architecture that dramatically improves analysis speed. We end with experimental results in Section 6.9.

6.1 Motivation and Examples

Below we show a typical memory leak found in C code:

```
p = malloc(...); ...
if (error_condition) return NULL;
return p;
```

Here, the programmer allocates a memory block memory and stores the reference in p. Under normal conditions p is returned to the caller, but in case of an error, the function returns NULL and the new location is leaked. The problem is fixed by inserting the statement free(p) immediately before the error return.

Our goal is to find these errors automatically. We note that leaks are always a flow-sensitive property, but sometimes are path-sensitive as well. The following example shows a common usage where a memory block is freed when its reference is non-NULL.

```
if (p != NULL)
    free(p);
```

To avoid false positives in their path insensitive leak detector, Heine *et. al.* [Heine and Lam 2003] transform this code into:

```
if (p != NULL)
    free(p);
else
    p = NULL;
```

The transformation handles the idiom with a slight change of program semantics (i.e., the extra NULL assignment to p). However, syntactic manipulations are unlikely to succeed in more complicated examples:

```
char fastbuf[10], *p;
if (len < 10)
    p = fastbuf;
else
    p = (char *)malloc(len);
...
if (p != fastbuf)
    free(p);
```

In this case, depending on the length of the required buffer, the programmer chooses between a smaller but more efficient stack-allocated buffer and a larger but slower heap-allocated one. This optimization is common in performance critical code such as Samba and the Linux kernel and a fully path sensitive analysis is desirable in analyzing such code.

Another challenge to the analysis is illustrated by the following example:

```
p−>name = strdup(string);
push_on_stack(p);
```

To correctly analyze this code, the analysis must infer that strdup allocates new memory and that push_on_stack adds an external reference to its first argument p and therefore causes (*p).name to escape. Thus, an interprocedural analysis is required. Without abstraction, interprocedural program analysis is prohibitively expensive for path sensitive analyses such as ours. As with the lock checker, we use a summary-based approach that exploits the natural abstraction boundary at function calls. For each function, we use SAT queries to infer information about the function's memory behavior and construct a summary for that function. The summary is designed to capture the following two properties:

(1) whether the function is a memory allocator, and
(2) the set of escaping objects that are reachable from the function's parameters.

We show how we infer and use such function summaries in Section 6.5.

6.2 Outline of the Leak Checker

This subsection discusses several key ideas behind the leak checker. First of all, we observe that pointers are not all equal with respect to memory leaks. Consider the following example:

```
(*p).data = malloc(...);
return;
```

The code contains a leak if p is a local variable, but not if p is a global or a parameter. In the case where *p itself is newly allocated in the current procedure, (*p).data escapes only if object *p escapes (except for cases involving cyclic structures; see below). In order to distinguish between these cases, we need a concept called *access paths* (Section 6.3) to track the paths through which objects are accessed from both inside and outside (if possible) the function body. We describe details about how we model object accessibility in Section 6.4.

References to a new memory location can also escape through means other than pointer references:

(1) memory blocks may be deallocated;
(2) function calls may create external references to newly allocated locations;
(3) references can be transferred via program constructs in C that currently are not modeled in Saturn (e.g., by decomposing a pointer into a page number and a page offset, and reconstructing it later).

To model these cases, we instrument every allocated memory block with a boolean escape attribute whose default value is false. We set the escape attribute to true whenever we encounter one of these three situations. A memory block is not considered leaked when its escape attribute is set.

$$
\begin{aligned}
\mathsf{Params} &= \{\mathsf{param}_0, \ldots, \mathsf{param}_{n-1}\} \\
\mathsf{Origins}\ (r) &::= \{\mathsf{ret\_val}\}\ \cup \mathsf{Params} \cup \\
&\qquad \mathsf{Globals} \cup \mathsf{NewLocs} \cup \mathsf{Locals} \\
\mathsf{AccPath}\ (\pi) &::= r \mid \pi.f \mid\ * \pi
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{PathOf} &: \mathsf{Loc} \rightarrow \mathsf{AccPath} \\
\mathsf{RootOf} &: \mathsf{AccPath} \rightarrow \mathsf{Origins}
\end{aligned}
$$

Fig. 13. Access paths.

One final issue that requires explicit modeling is that malloc functions in C might fail. When it does, malloc returns null to signal a failed allocation. This situation is illustrated in Section 6.1 and requires special-case handling in path insensitive analyses. We use a boolean valid attribute to track the return status of each memory allocation. The attribute is non-deterministically set at each allocation site to model both success and failure scenarios. For a leak to occur, the corresponding allocation must originate from a successful allocation and thus have its valid attribute set to true.

6.3 Access Paths and Origins

This subsection extends the interface object concept introduced in Section 5.1 to track and manipulate the path through which objects are first accessed. Following standard literature on alias and escape analysis, we call the revised definition *access paths*. As shown in the Section 6.2, access path information is important in defining the escape condition for memory locations.

Figure 13 defines the representation and operations on access paths, which are interface objects (see Section 5.1) extended with Locals and NewLocs. Objects are reached by field accesses or pointer dereferences from five origins: global and local variables, the return value, function parameters, and newly allocated memory locations. We represent the path through which an object is accessed first with AccPath.

PathOf maps objects (and polymorphic locations) to their access paths and access path information is computed by recording object access paths used during the analysis. The RootOf function takes an access path and returns the object from which the path originates.

We illustrate these concepts using the following example:

```
struct state { void *data; };
void *g;
void f(struct state *p)
{
  int *q;
  g = p−>data;
  q = g;
  return q; /* rv = q */
}
```

Table V summarizes the objects reached by the function, their access paths and origins. The origin and path information indicates how these objects are first accessed and is used in defining the leak conditions in Section 6.4.

Object	AccPath	RootOf
p	param_0	param_0
$*p$	$*\mathsf{param}_0$	param_0
$(*p).data$	$(*\mathsf{param}_0).data$	param_0
$*(*p).data$	$*(*\mathsf{param}_0).data$	param_0
g	global_g	global_g
q	local_q	local_q
rv	ret_val	ret_val

Table V. Objects, access paths, and access origins in the sample program.

6.4 Escape and Leak Conditions

$$\frac{\psi(p) = \{\!|\ (\mathcal{G}_0, l_0), \ldots, (\mathcal{G}_{n-1}, l_{n-1})\ |\!\}}{\mathsf{PointsTo}(p,l) = \begin{cases} \mathcal{G}_i & \text{if } \exists i \text{ s.t. } \mathsf{AddrOf}(l) {=} \mathsf{AddrOf}(l_i) \\ \mathsf{false} & \text{otherwise} \end{cases}} \qquad \textsf{points-to}$$

$$\textsf{Excluded Set:} \quad \mathcal{X} \subseteq \mathsf{Origins} - (\mathsf{Globals} \cup \mathsf{Locals})$$

$$\frac{\mathsf{RootOf}(p) \in \mathsf{Locals} \cup \mathcal{X}}{\mathsf{EscapeVia}(l, p, \mathcal{X}) = \mathsf{false}} \qquad \textsf{via-local}$$

$$\frac{\mathsf{RootOf}(p) \in \mathsf{Globals}}{\mathsf{EscapeVia}(l, p, \mathcal{X}) = \mathsf{PointsTo}(p, l)} \qquad \textsf{via-global}$$

$$\frac{\mathsf{RootOf}(p) = (\mathsf{Params} \cup \{\mathsf{ret\_val}\}) - \mathcal{X}}{\mathsf{EscapeVia}(l, p, \mathcal{X}) = \mathsf{PointsTo}(p, l)} \qquad \textsf{via-interface}$$

$$\frac{l' = \mathsf{RootOf}(p) \quad l' \in (\mathsf{NewLocs} - \mathcal{X})}{\mathsf{EscapeVia}(l,\ p, \mathcal{X}) = \mathsf{PointsTo}(p, l) \wedge \mathsf{Escaped}(l', \mathcal{X} \cup \{l\})} \qquad \textsf{via-newloc}$$

$$\mathsf{Escaped}(l, \mathcal{X}) = [\![l\#\mathsf{escaped}]\!]_\psi \vee \bigvee\nolimits_p \mathsf{EscapeVia}(l, p, \mathcal{X}) \qquad \textsf{escaped}$$

$$\mathsf{Leaked}(l, \mathcal{X}) = [\![l\#\mathsf{valid}]\!]_\psi \wedge \neg \mathsf{Escaped}(l, \mathcal{X}) \qquad \textsf{leaked}$$

*For brevity, $\mathsf{RootOf}(p)$ denotes $\mathsf{RootOf}(\mathsf{PathOf}(p))$.

Fig. 14. Memory leak detection rules.

Figure 14 defines the rules we use to find memory leaks and construct function summaries. As discussed in Section 5.4, we assume that there is one unique exit block in each function's control flow graph. We apply the leak rules at the end of the exit block, and the implicitly defined environment ψ in the rules refers to the exit environment.

In Figure 14, the $\mathsf{PointsTo}(p, l)$ function gives the condition under which pointer p points to location l. The result is simply the guard associated with l if it occurs in the GLS of p and false otherwise. Using the $\mathsf{PointsTo}$ function, we are ready to define the escape relationships $\mathsf{Escaped}$ and $\mathsf{EscapeVia}$.

$$\begin{array}{l}\mathsf{Escapee}(\epsilon) ::= \mathsf{param}_i \mid \epsilon.f \mid *\epsilon \\ \mathsf{Summary} : \Sigma \in \mathsf{bool} \times 2^{\mathsf{Escapee}}\end{array}$$

Fig. 15. The definition of function summaries.

Ignoring the exclusion set $\mathcal{X}$ for now, $\mathsf{EscapeVia}(l, p, \mathcal{X})$ returns the condition under which location l escapes through pointer p. Depending on the origin of p, EscapeVia is defined by four rules via-* in Figure 14. The simplest of the four rules is via-local, which stipulates that location l cannot escape through p if p's origin is a local variable, since the reference is lost when p goes out of scope at function exit.

The rule via-global handles the case where p is accessible through a global variable. In this case, l escapes when p points to l, which is described by the condition $\mathsf{PointsTo}(p, l)$. The case where a location escapes through a function parameter is treated similarly in the via-interface rule.

The rule via-newloc handles the case where p is a newly allocated location. Again ignoring the exclusion set $\mathcal{X}$, the rule stipulates that a location l escapes if p points to l and the origin of p, which is itself a new location, in turn escapes.

However, the above statement is overly generous in the following situation:

```
s = malloc(...); /* creates new location l' */
s–>next = malloc(...);          /* creates l */
s–>next–>prev = s;  /* circular reference */
```

The circular dependency that l escapes if l' does, and vice versa, can be satisfied by the constraint solver by assuming both locations escape. To find this leak, we prefer a solution where neither escapes. We solve this problem by adding an *exclusion set* $\mathcal{X}$ to the leak rules to prevent circular escape routes. In the via-newloc rule, the location l in question is added to the exclusion set, which prevents l' from escaping through l.

The $\mathsf{Escaped}(l, \mathcal{X})$ function used by the via-newloc rule computes the condition under which l escapes through a route that does not intersect with $\mathcal{X}$. It is defined by considering escape routes through all pointers and other means such as function calls (modeled by the attribute $l\#\mathsf{escaped}$).

Finally, $\mathsf{Leaked}(l, \mathcal{X})$ computes the condition under which a new location l is leaked through some route that does not intersect with $\mathcal{X}$. It takes into consideration the *validity* of l, which models whether the initial allocation is successful or not (see Section 6.1 for an example).

Using these definitions, we specify the condition under which a leak error occurs:

$$\exists l \text{ s.t. } (l \in \mathsf{NewLocs}) \text{ and } (\mathsf{Leaked}(l, \{\}) \text{ is satisfiable})$$

We issue a warning for each location that satisfies this condition.

6.5 Interprocedural Analysis

This subsection describes the summary-based approach to interprocedural leak detection in Saturn. We start by defining the summary representation in Section 6.5.1 and discuss summary generation and application in Sections 6.5.2 and 6.5.3.

6.5.1 *Summary Representation.* Figure 15 shows the representation of a function summary. In leak analysis we are interested in whether the function returns

$\boxed{\text{IsMalloc:}}$

$$\psi(\mathsf{rv}) = \{\!|\ (\mathcal{G}_0, \mathsf{null}), \overline{(\mathcal{G}_i, l_i)}, \overline{(\mathcal{G}'_j, l'_j)}\ |\!\}$$
$$\text{where } l_i \in \mathsf{NewLocs} \text{ and } l'_j \notin \mathsf{NewLocs}$$

- $\bigvee_i \mathcal{G}_i$ is satisfiable and $\bigvee_j \mathcal{G}'_j$ is not satisfiable
- $\forall l_i \in \mathsf{NewLocs}, (\mathcal{G}_i \implies \mathsf{Leaked}(l_i, \{\mathsf{ret\_val}\}))$ is a tautology

$\boxed{\text{Escapees:}}$

$$\mathsf{EscapedSet}(f) = \{\ \mathsf{PathOf}(l) \mid \mathsf{RootOf}(l) = \mathsf{param}_i \text{ and } \mathsf{Escaped}(l, \{\mathsf{param}_i\}) \text{ is satisfiable }\}$$

Fig. 16. Summary generation.

newly allocated memory (i.e. allocator functions), and whether it creates any external reference to objects passed via parameters (recall Section 6.1). Therefore, a summary Σ is composed of two components: 1) a boolean value that describes whether the function returns newly allocated memory, and 2) a set of escaped locations (*escapees*). Since caller and callee have different names for the formal and actual parameters, we use access paths (recall Section 6.3) to name escaped objects. These paths, called Escapees in Figure 15, are defined as a subset of access paths whose origin is a parameter.

Consider the following example:

```
1 void *global;
2 void *f(struct state *p) {
3   global = p->next->data;
4   return malloc(5);
5 }
```

The summary for function f is computed as

$$\langle \text{isMalloc: true; escapees: } \{(*(*\mathsf{param}_0).\mathsf{next}).\mathsf{data}\}\rangle$$

because f returns newly allocated memory at line 4 and adds a reference to p->next->data from global and therefore escapes that object.

Notice that the summary representation focuses on common leak scenarios. It does not capture all memory allocations. For example, functions that return new memory blocks via a parameter (instead of the return value) are not considered allocators. Likewise, aliasing relationships between parameters are not captured by the summary representation.

6.5.2 *Summary Generation.* Figure 16 describes the rules for function summary generation. When the return value of a function is a pointer, the IsMalloc rule is used to decide whether a function returns a newly allocated memory block. A function qualifies as a *memory allocator* if it meets the following two conditions:

(1) The return value can only point to null or newly allocated memory locations. The possibility of returning any other existing locations disqualifies the function as a memory allocator.

(2) The return value is the only externally visible reference to new locations that might be returned. This prevents false positives from region-based memory management schemes where a reference is retained by the allocator to free all new locations in a region together.

The set of escaped locations is computed by iterating through all parameter accessible objects (i.e., objects whose access path origin is a parameter p), and testing whether the object can escape through a route that does not go through p, i.e., if $\mathsf{Escaped}(l, \{\mathsf{param}_i\})$ is satisfiable.

Take the following code as an example:

```
void insert_after(struct node *head, struct node *new) {
  new->next = head->next;
  head->next = new;
}
```

The escapee set of insert_after includes: (*head).next, since it can be reached by the pointer (*new).next; and *new, since it can be reached by the pointer (*head).next. The object *head is not included, because it is only accessible through the pointer head, which is excluded as a possible escape route. (For clarity, we use the more mnemonic names head and next instead of param_0 and param_1 in these access paths.)

6.5.3 *Summary Application.* Function calls are replaced by code that simulates their memory behavior based on their summary. The following pseudo-code models the effect of the function call r = f(e_1, e_2, ..., e_n), assuming f is an allocator function with escapee set escapees:

```
1   /* escape the escapees */
2   foreach (e) in escapees do
3       (*e)#escaped = true;
4
5   /* allocate new memory, and store it in r */
6   if (*) {
7       newloc(r);
8       (*r)#valid <- true;
9   } else
10      r <- null;
```

Lines 1-3 set the escaped attribute for f's escapees. Note that e at line 3 is an access path from a parameter. Thus (*e) is not strictly a valid Saturn object and must be transformed into one using a series of assignments. The details are omitted for brevity.

Lines 5-10 simulate the memory allocation performed by f. We non-deterministically assign a new location to r and set the valid bit of the new object to true. To simulate a failed allocation, we assign null to r at line 10.

In the case where f is not an allocation function, lines 5-10 are replaced by the statement r ← unknown.

6.6 Loops and Recursion

For the leak detector Saturn uses a two-pass algorithm for loops. In the first pass the loop is unrolled a small number of times (three in our implementation) and the backedges discarded; thus, just the first three iterations are analyzed. For leak detection this strategy works well except for loops such as

```
for (i = 0; i < 10000; i++)
    ;
```

The problem here is that the loop exit condition is never true in the first few iterations of the loop. Thus path sensitive analysis of just the first few iterations concludes that the exit test is never satisfied and the code after the loop appears to be unreachable. In the first pass, if the loop can terminate within the number of unrolled iterations, the analysis of the loop is just the result of the first pass. Otherwise, we discard results from the first pass and a second, more conservative analysis is used. In the second pass, we replace the right-hand side of all assignments in the loop body by `unknown` expressions and the loop is analyzed once (havoc'ing). Intuitively, the second pass analyzes the last iteration of the loop; we model the fact that we do not know the state of modified variables after an arbitrary number of earlier iterations by assigning them `unknown` values [Xie and Chou 2002]. Integer `unknown`s are represented using unconstrained Boolean variables, and are thus conservative approximations of the actual runtime values. For pointers, however, `unknown` currently evaluates to a fresh location, and therefore is unsound. The motivation for this two-pass analysis is that the first pass yields more precise results when the loop can be shown to terminate; however, if the unrolled loop iterations cannot reach the loop exit, then the second pass is preferable because it is more important to at least reach the code after the loop than to have precise information for the loop itself.

Recursion is handled in a similar manner as in the Linux lock checker: we analyze mutually recursive functions once in arbitrary order. Theoretically, this is a source of both false positives (missed escape routes through recursive function calls) and false negatives (missed allocation functions). Practically, however, the loss of precision due to recursion is minimal in our experiments.

6.7 Handling Unsafe Operations in C

The C type system allows constructs (i.e., unsafe type casts and pointer arithmetic) not currently modeled by Saturn. We have identified several common idioms that use such operations, motivating some extensions to our leak detector.

One extension handles cases similar to the following, which emulates a form of inheritance in C:

```
struct sub { int value; struct super super; }
struct super *allocator(int size)
{
    struct sub *p = malloc(...);
    p->value = ...;
    return (&p->super);
}
```

The allocator function returns a reference to the super field of the newly allocated memory block. Technically, the reference to sub is lost on exit, but it is not considered an error because it can be recovered with pointer arithmetic. Variants of this idiom occur frequently in the projects we examined. Our solution is to consider a structure escaped if any of its components escape.

Another extension recognizes common address manipulation macros in Linux such as virt_to_phys and bus_to_virt, which add or subtract a constant page offset to arrive at the physical or virtual equivalent of the input address. Our implementation matches such operations and treats them as identity functions.

6.8 A Distributed Architecture

The leak analysis uses a path sensitive analysis to track every incoming and newly allocated memory location in a function. Compared to the lock checker in Section 5, the higher number of tracked objects (and thus SAT queries) means the leak analysis is much more computationally intensive.

However, Saturn is highly parallelizable, because it analyzes each function separately, subject only to the ordering dependencies of the function call graph. We have implemented a distributed client/server architecture to exploit this parallelism in the memory leak checker.

The server side consists of a scheduler, dispatcher, and database server. The scheduler computes the dependence graph between functions and determines the set of functions ready to be analyzed. The dispatcher sends ready tasks to idle clients. When the client receives a new task, it retrieves the function's abstract syntax tree and summaries of its callees from the database server. The result of the analysis is a new summary for the analyzed function, which is sent to the database server for use by the function's callers.

We employ caching techniques to avoid congestion at the server. Our implementation scales to hundreds of CPUs and is highly effective: the analysis time for the Linux kernel, which requires nearly 24 hours on a single fast machine, is analyzed in 50 minutes using around 80 unloaded CPUs.[8] The speedup is sublinear in the number of processors because there is not always enough parallelism to keep all processors busy, particularly near the root of a call graph.

Due to the similarity of the analysis architecture between the Linux lock checker and the memory leak detector, we expect that the former would also benefit from a distributed implementation and achieve similar speed up.

6.9 Experimental Results

We have implemented the leak checker as a plug-in to the Saturn analysis framework and applied it to five user space applications and the Linux kernel.

6.9.1 *User Space Applications.* We checked five user space software packages: Samba, OpenSSL, PostFix, Binutils, and OpenSSH. We analyzed the latest release of the first three, while we used older versions of the last two to compare with results reported for other leak detectors [Heine and Lam 2003; Hackett and Rugina 2005]. All experiments were done on a lightly loaded dual Xeon™ 2.8G server with 4 gigabytes of memory as well as on a heterogeneous cluster of around 80 idle

[8] As courtesy to the generous owners of these machines, we constantly monitor CPU load and user activity on these machines, and turn off clients that have active users or tasks. Furthermore, these 80 CPUs range from low-end Pentium 4 1.8G workstations to high-end Xeon 2.8G servers in dual- and quad-processor configurations. Thus, performance statistics for distributed runs reported here only provide an approximate notion of speed-up when compared to single processor analysis runs.

	LOC	**Single Proc.**		**Distributed**	
		Time	**LOC/s**	**P.Time**	**P.LOC/s**
User-space App.					
Samba	403,744	3h22m52s	33	10m57s	615
OpenSSL	296,192	3h33m41s	23	11m09s	443
Postfix	137,091	1h22m04s	28	12m00s	190
Binutils	909,476	4h00m11s	63	16m37s	912
OpenSSH	36,676	27m34s	22	6m00s	102
Sub-total	1,783,179	12h46m22s	39	56m43s	524
Linux Kernel					
v2.6.10	5,039,296	23h13m27s	60	50m34s	1661
Total	6,822,475	35h59m49s	53	1h47m17s	1060

LOC: total number of lines of code; **Time**: analysis time on a single processor (2.8G Xeon);
P.Time: parallel analysis time on a heterogeneous cluster of around 80 unloaded CPUs.

(a) Performance Statistics.

	Fn	**Failed** (%)	**Alloc**	**Bugs**	**FP** (%)
User-space App.					
Samba	7,432	24 (0.3%)	80	83	8 (8.79%)
OpenSSL	4,181	60 (1.4%)	101	117	1 (0.85%)
Postfix	1,589	11 (0.7%)	96	8	0 (0%)
Binutils	2,982	36 (1.2%)	91	136	5 (3.55%)
OpenSSH	607	5 (0.8%)	19	29	0 (0%)
Sub-total	16,791	136 (0.8%)	387	373	14 (3.62%)
Linux Kernel					
v2.6.10	74,367	792 (1.1%)	368	82	41 (33%)
Total	91,158	928 (1.0%)	755	455	55 (10.8%)

Fn: number of functions in the program;
Alloc: number of memory allocators detected;
FP: number of false positives.

(b) Analysis results.

Table VI. Experimental results for the memory leak checker.

workstations. For each function, the resource limits were set to 512MB of memory and 90 seconds of CPU time.

The top portions of Tables VI(a) and (b) give the performance statistics and bug counts of the leak checker on the five user-space applications. Note that we miss any bugs in the small percentage of functions where resource limits are exceeded. The 1.8 million lines of code were analyzed in under 13 hours using a single processor and in under 1 hour using a cluster of about 80 CPUs. The parallel speedups increase significantly with project size, indicating larger projects have relatively fewer call graph dependencies than small projects. Note that the sequential scaling behavior (measured in lines of code per second) remains stable across projects ranging from 36K up to 909K lines of unpreprocessed code.

The tool issued 379 warnings across these applications. We have examined all the warnings and believe 365 of them are bugs. (Warnings are per allocation site to facilitate inspection.) Besides bug reports, the leak checker generates a database of function summaries documenting each function's memory behavior. In our ex-

```
/* Samba – libads/ldap.c:ads_leave_realm */
host = strdup(hostname);
if (. . .) {
  . . .;
  return ADS_ERROR_SYSTEM(ENOENT);
}
. . .
```

(a) The programmer forgot to free host on an error exit path.

```
/* Samba – client/clitar.c:do_tarput */
longfilename = get_longfilename(finfo);
. . .
return;
```

(b) The programmer apparently is not aware that get_longfilename allocates new memory, and forgets to de-allocate longfilename on exit.

```
/* Samba – utils/net_rpc.c:rpc_trustdom_revoke */
domain_name = smb_xstrdup(argv[0]);
. . .
if (!trusted_domain_password_delete(domain_name))
     . . .
return . .;
```

(c) trusted_domain_password_delete does not de-allocate memory, as its name might suggest. Memory referenced by domain_name is thus leaked on exit.

Fig. 17. Three representative errors found by the leak checker.

```
/* OpenSSL – crypto/bn/bn_lib.c:BN_copy */
t = BN_new();
if (t == NULL) return (NULL);
r = BN_copy(t, a);
if (r == NULL)
    BN_free(t);
return r;
```

Fig. 18. A sample false positive.

perience, the function summaries are highly accurate, and that, combined with path-sensitive intraprocedural analysis, explains the exceptionally low false positive rate. The summary database's function level granularity enabled us to focus on one function at a time during inspection, which facilitated bug confirmation.

Most of the bugs we found can be classified into three main categories:

(1) *Missed deallocation on error paths.* This case is by far the most common, often happening when the procedure has multiple allocation sites and error conditions. Errors are common even when the programmer has made an effort to clean-up orphaned memory blocks. Figure 17a gives an example.

(2) *Missed allocators.* Not all memory allocators have names like OPENSSL_malloc. Programmers sometimes forget to free results from less obvious allocators such as get_longfilename (`samba/client/clitar.c`, Figure 17b).

(3) *Non-escaping procedure calls.* Despite the suggestive name, trusted_domain_password_delete (`samba/passdb/secrets.c`) does not free its parameter (Figure 17c).

Figure 18 shows a false positive caused by a limitation of our choice of function summaries. At line 4, BN_copy returns a copy of t on success and null on failure, which is not detected, nor is it expressible by the function summary.

6.9.2 *The Linux Kernel.* The bottom portions of Tables VI(a) and (b) summarize statistics of our experiments on Linux 2.6.10. Using the parallel analysis framework (recall Section 6.8) we distributed the analysis workload on 80 CPUs. The analysis completed in 50 minutes, processing 1661 lines per second. We are not aware of any other analysis algorithm that achieves this level of parallelism.

The bug count for Linux is considerably lower than for the other applications relative to the size of the source code. The Linux project has made a conscious effort to reduce memory leaks, and, in most cases, they try to recover from error conditions, where most of the leaks occur. Nevertheless, the tool found 82 leak errors, some of which were surrounded by error handling code that frees a number of other resources. Two errors were confirmed by the developers as exploitable and could potentially enable denial of service attacks against the system. These bugs were immediately fixed when reported.

The false positive rate is higher in the kernel than user space applications due to wide-spread use of function pointers and pointer arithmetic. Of the 41 false positives, 16 are due to calls via function pointers and 9 due to pointer arithmetic. Application specific logic accounted for another 12, and the remaining 4 are due to Saturn's current limitations in modeling constructs such as arrays and unions.

7. UNSOUNDNESS

One theoretical weakness of the two checkers, as described above, is unsoundness. In this section, we briefly summarize the sources of unsoundness. Both the finite-state machine (FSM) checker and the memory leak analysis share the following sources of unsoundness:

(1) **Handling of loops.** We introduced two techniques to handle loops in Saturn: unrolling and havoc'ing, both of which are unsound. The former might miss bugs that occur only in a long-running loop, and the latter is unsound in its treatment of modified pointers in the loop body (see Section 6.6).

(2) **Handling of recursion.** Recursive function calls are not handled in the two checkers, so bugs could remain undetected due to inaccurate function summaries.

(3) **Interprocedural aliasing.** Both checkers use the heuristic that distinct pointers from the external environment (e.g. function parameters, global variables) point to distinct objects. Although effective in practice, this heuristic may prevent our analysis from detecting bugs caused by interprocedural aliasing.

(4) **Summary representation.** The function summary representations for both checkers leave several aspects of a function's behavior unspecified. Examples

include interprocedural side-effects (e.g. modification of global variables) and aliasing, both of which may lead to false negatives.

(5) **Unhandled C constructs.** For efficiency reasons, constructs such as unions, arrays, and pointer arithmetic are not directly modeled by the Saturn framework. Rather, they are handled by specific checkers during translation from C to the Saturn intermediate language. For example, in the leak checker, memory blocks stored in arrays are considered to be escaped, which is a source of unsoundness.

It is worth noting that unsoundness is not a fundamental limitation of the Saturn framework. Sound analyses can be constructed in Saturn by using appropriate summaries for both loops and functions and by iterating the analyses to reach a fixed point. For example, [Hackett and Aiken 2005] describes the design and implementation of a sound and precise pointer alias analysis in Saturn.

8. RELATED WORK

In this section we discuss the relationship of Saturn to several other systems for error detection and program verification.

8.1 FSM Checking

Several previous systems have been successfully applied to checking finite state machine properties in system code. Saturn was partly inspired by the first author's previous work on *Meta Compilation* (MC) [Engler et al. 2000; Hallem et al. 2002] 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 [Foster et al. 2002], 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 relevant `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,

```
/* 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 [Engler et al. 2000]). 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 [Ball and Rajamani 2001; Henzinger et al. 2003; Henzinger et al. 2002], these are whole-program analyses. ESP, a lower-complexity approach based on context-free reachability, is similarly whole-program [Das et al. 2002]. In contrast, Saturn analyzes open programs and computes summaries for functions independent of their calling context. In our experiments, Saturn 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.

CQual is a quite different, type-based approach to program checking [Foster et al. 2002; Aiken et al. 2003]. 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.

8.2 Memory Leak Detection

Memory leak detection using dynamic tools has been a standard part of the working programmer's toolkit for more than a decade. One of the earliest and best known tools is *Purify* [Hastings and Joyce 1992]; see [Chilimbi and Hauswirth 2004] for a recent and significantly different approach to dynamic leak detection. Dynamic memory leak detection is limited by the quality of the test suite; unless a test case triggers the memory leak it cannot be found.

More recently there has been work on detecting memory leaks statically, sometimes as an application of general shape or heap analysis techniques, but in other cases focusing on leak detection as an interesting program analysis problem in its own right. One of the earliest static leak detectors was LCLint [Evans 1996], which employs an intraprocedural dataflow analysis to find likely memory errors. The analysis depends heavily on user annotation to model function calls, thus requiring substantial manual effort to use. The reported false positive rate is high mainly due to path insensitive analysis.

Prefix [Bush et al. 2000] detects memory leaks by symbolic simulation. Like Saturn, Prefix uses function summaries for scalability and is path sensitive. However, Prefix explicitly explores paths one at a time, which is expensive for procedures with many paths. Heuristics limit the search to a small set of "interesting" paths. In contrast, Saturn represents all paths using boolean constraints and path exploration is implicit as part of boolean constraint solving.

Chou [Chou 2003] describes a path-sensitive leak detection system based on static reference counting. If the static reference count (which over-approximates the dynamic reference count) becomes zero for an object that has not escaped, that object is leaked. Chou reports finding hundreds of memory leaks in an earlier Linux kernel using this method, most of which have since been patched. The analysis is quite conservative in what it considers escaping; for example, saving an address in the heap or passing it as a function argument both cause the analysis to treat the memory at that address as escaped (i.e., not leaked). The interprocedural aspect of the analysis is a conservative test to discover `malloc` wrappers. Saturn's path- and context-sensitive analysis is more precise both intra- and inter-procedurally.

We know of two memory leak analyses that are sound and for which substantial experimental data is available. Heine and Lam use *ownership types* to track an object's owning reference (the reference responsible for deallocating the object) [Heine and Lam 2003]. Hackett and Rugina describe a hybrid region and shape analysis (where the regions are given by the equivalence classes defined by an underlying points-to analysis) [Hackett and Rugina 2005]. In both cases, on the same inputs Saturn finds more bugs with a lower false positive rate. While Saturn's lower false positive is not surprising (soundness usually comes at the expense of more false positives), the higher bug counts for Saturn are surprising (because sound tools should not miss any bugs). For example, for `binutils` Saturn found 136 bugs compared with 66 found by Heine and Lam. The reason appears to be that Heine and Lam inspected only 279 of 1106 warnings generated by their system; the other 727 warnings were considered likely to be false positives. (Saturn did miss one bug reported by Heine and Lam due to exceeding the CPU time limit for the function containing the bug.) Hackett and Rugina report 10 bugs in `OpenSSH` out of 26 warnings. Here there appear to be two issues. First, the abstraction for which the algorithm is sound does not model some common features of C, causing the implementation for C to miss some bugs. Second, the implementation does not always finish (just as Saturn does not).

There has been extensive prior research in points-to and escape analysis. Access paths were first used by Landi and Ryder [Landi and Ryder 1992] as symbolic names for memory locations accessed in a procedure. Several later algorithms (e.g., [Emami et al. 1994; Wilson and Lam 1995; Liang and Harrold 2001]) also make use of parameterized pointer information to achieve context sensitivity. Escape analysis (e.g. [Whaley and Rinard 1999; Ruf 2000]) determines the set of objects that do not escape a certain region. The result is traditionally used in program optimizers to remove unnecessary synchronization operations (for objects that never escape a thread) or enable stack allocation (for ones that never escape a function call). Leak detection benefits greatly from path-sensitivity, which is not a property of traditional escape analyses.

8.3 Other SAT-based Checking and Verification Tools

Rapid improvements in algorithms for SAT (e.g. zChaff [Zhang et al. 2001; Moskewicz et al. 2001], which we use in SATURN) have led to its use in a variety of applications, including recently in program verification.

Jackson and Vaziri were apparently the first to consider finding bugs via reducing program source to boolean formulas [Jackson and Vaziri 2000]. Subsequently there has been significant work on a similar approach called *bounded model checking* [Kroening et al. 2003]. In [Clarke et al. 2004], Clarke et. al. has further explored the idea of SAT-based predicate abstraction of ANSI-C programs. While there are many low-level 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 inference, as well as checking, of properties without separate programmer-written specifications.

9. 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 have experimentally validated our approach by conducting two case studies involving a Linux lock checker and a memory leak checker. Results from the experiments show that our system scales well, parallelizes well, and finds more errors with less false positives than previous error detection systems.

REFERENCES

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts.

AIKEN, A., FOSTER, J. S., KODUMAL, J., AND TERAUCHI, T. 2003. Checking and inferring local non-aliasing. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM Press, 129–140.

BALL, T., COOK, B., LEVIN, V., AND RAJAMANI, S. 2004. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *Proceedings of Fourth International Conference on Integrated Formal Methods.* Springer.

BALL, T. AND RAJAMANI, S. K. 2001. Automatically validating temporal safety properties of interfaces. In *Proceedings of SPIN 2001 Workshop on Model Checking of Software.* 103–122. LNCS 2057.

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

BUSH, W., PINCUS, J., AND SIELAFF, D. 2000. A static analyzer for finding dynamic programming errors. *Software—Practice & Experience 30,* 7 (June), 775–802.

CHILIMBI, T. AND HAUSWIRTH, M. 2004. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems.*

CHOU, A. 2003. Static analysis for bug finding in systems software. Ph.D. thesis, Stanford University.

CLARKE, E., KROENING, D., AND LERDA, F. 2004. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, K. Jensen and A. Podelski, Eds. Lecture Notes in Computer Science, vol. 2988. Springer, 168–176.

CLARKE, E., KROENING, D., SHARYGINA, N., AND YORAV, K. 2004. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in Systems Design 25,* 2-3 (Sept.), 105–127.

Das, M., Lerner, S., and Seigle, M. 2002. Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. Berlin, Germany.

Emami, M., Ghiya, R., and Hendren, L. 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*.

Engler, D., Chelf, B., Chou, A., and Hallem, S. 2000. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of Operating Systems Design and Implementation (OSDI)*.

Evans, D. 1996. Static detection of dynamic memory errors. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*.

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

Hackett, B. and Aiken, A. 2005. How is aliasing used in systems software? Tech. rep., Stanford University.

Hackett, B. and Rugina, R. 2005. Region-based shape analysis with tracked locations. In *Proceedings of the 32nd Annual Symposium on Principles of Programming Languages*.

Hallem, S., Chelf, B., Xie, Y., and Engler, D. 2002. 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.

Hastings, R. and Joyce, B. 1992. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*.

Heine, D. L. and Lam, M. S. 2003. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. 168–181.

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

Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G. 2003. Software verification with Blast. In *Proceedings of the SPIN 2003 Workshop on Model Checking Software*. 235–239. LNCS 2648.

Jackson, D. and Vaziri, M. 2000. Finding bugs with a constraint solver. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*.

Khurshid, S., Pasareanu, C., and Visser, W. 2003. 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.

Kroening, D., Clarke, E., and Yorav, K. 2003. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of the 40th Design Automation Conference*. ACM Press, 368–371.

Landi, W. and Ryder, B. 1992. A safe approximation algorithm for interprocedural pointer aliasing. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*.

Liang, D. and Harrold, M. 2001. Efficient computation of parameterized pointer information for interprocedural analysis. In *Proceedings of the 8th Static Analysis Symposium*.

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

Ruf, E. 2000. Effective synchronization removal for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*.

Whaley, J. and Rinard, M. 1999. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*.

Wilson, R. and Lam, M. 1995. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*.

Xie, Y. and Chou, A. 2002. Path sensitive analysis using boolean satisfiability. Tech. rep., Stanford University. Nov.

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