Context- and Path-sensitive Memory Leak Detection*

Yichen Xie Alex Aiken

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

ABSTRACT

We present a context- and path-sensitive algorithm for detecting memory leaks in programs with explicit memory management. Our leak detection algorithm is based on an underlying escape analysis: any allocated location in a procedure P that is not deallocated in P and does not escape from P is leaked. We achieve very precise context- and path-sensitivity by expressing our analysis using boolean constraints. In experiments with six large open source projects our analysis produced 510 warnings of which 455 were unique memory leaks, a false positive rate of only 10.8%. A parallel implementation improves performance by over an order of magnitude on large projects; over five million lines of code in the Linux kernel is analyzed in 50 minutes.

Categories and Subject Descriptors

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

General Terms

Algorithms, Experimentation, Languages, Verification.

Keywords

Program analysis, error detection, memory management, memory leaks, boolean satisfiability.

1. INTRODUCTION

Languages with explicit memory management require the programmer to manually deallocate memory blocks that are no longer needed by the program, and memory leaks are a common problem in code written in such languages. A memory leak is particularly serious in long running applications, where it slowly consumes available memory, causing performance degradation and eventually crashing the system. Furthermore, a memory leak is among the hardest bugs to detect since it has few visible symptoms other than the slow and steady increase in memory consumption. Memory leaks remain an important problem in the development of many widely-used system programs. We make the following contributions to the problem of detecting memory leaks:

- We give a new algorithm for context-sensitive and path-sensitive escape analysis of dynamically allocated objects (Section 4). Our approach, based on representing computations as boolean constraints, is significantly more accurate than traditional approaches based on standard types or dataflow analysis.

- We present experimental results showing that our approach scales well and that the extra precision of using boolean constraints translates into static detection of many memory leaks with a low false positive rate (Section 5). Across a set of large C codebases our leak checker generates 510 warnings, of which 455 are distinct memory leaks[1]. The 55 false warnings represent a false positive rate of only 10.8%. Our approach finds many more leaks than previous approaches, including algorithms designed to be sound (see Section 6 for a discussion). The false positive rate of our algorithm is also much lower than that of any other static leak detection system known to us.

- Our memory leak checker is computationally intensive. Analysis of more than 5 MLOC in the Linux kernel, for example, currently requires one CPU-day on a fast CPU. However, our algorithm analyzes each function separately and, potentially, in parallel, subject only to the ordering dependencies of the function call graph. Exploiting this parallelism dramatically reduces analysis times: using roughly 80 unloaded CPUs the Linux kernel is checked in 50 minutes.

- SAT-based analysis models each bit of a computation as a separate boolean variable; for example, a 32-bit integer is represented by 32 boolean variables. A consequence is that static type information is required to know the size (and therefore the number of bits) of the values being modeled. We introduce the notion of a *polymorphic location*, a memory cell that may be used

*Supported by NSF grant CCF-0430378.

ESEC-FSE'05, September 5–9, 2005, Lisbon, Portugal.

[1]Bug reports are available online at `http://glide.stanford.edu/saturn/leaks/`.

as multiple types (Section 3). This feature is critical to accurate modeling of heap references in the presence of type casts; in object-oriented languages the same issue would arise with subtyping.

We wish to emphasize that our goal is a bug finding tool, not a verification tool: our method is unsound. As we will discuss, the main sources of unsoundness are in the handling of loops and interprocedural aliasing.

2. BACKGROUND

Our leak checker is implemented in the SATURN analysis framework [18]. This section briefly describes how SATURN models the scalar portion of a simple procedural language, including base types (integers), structures, and the arbitrary control-flow found in C. Handling pointers in a memory leak checker requires new ideas and is discussed in Section 3.

Figure 1 defines the scalar entities SATURN manipulates. A *type* is either an integer type or a structure type. An integer type (n, s) includes the number of bits n and whether the integer is signed or unsigned. Such types are represented (see the end of Figure 1) by a vector of boolean formulas (which may include boolean variables) of length n. Structure types are simply records with named fields.

Programs consist of objects, expressions, conditions, and statements. An *object* is either a program variable or a structure. Objects, a few basic constructs (constants, and unary and binary operations), and type casts $(\tau)e$ require no further explanation. A `lift` expression converts a condition (see below) to an expression of the appropriate type. Because conditions are single bits, this operation has the effect of padding the condition to the number of bits for the type. An `unknown` expression is used to mark features of C that we do not model (e.g., division); an $\texttt{unknown}(\tau)$ expression is represented by a vector of unconstrained boolean variables of the length given by τ.

Expressions are translated to boolean formulas, defined by cases using rules of the form

$$[\![e]\!]_\psi = \beta,$$

where e ranges over expressions, and β over boolean representations. The environment $\psi : v \mapsto \beta$ gives the boolean representation of the free variables of e. As an example, the following rule describes the bitwise-and operation, which evaluates the two operands e and e' in the current environment ψ and constructs the result by building a boolean representation for the bitwise-and of both operands.

$$\frac{[\![e]\!]_\psi = [b_{n-1} \ldots b_0]_s \quad [\![e']\!]_\psi = [b'_{n-1} \ldots b'_0]_s}{[\![e \texttt{ band } e']\!]_\psi = [b_{n-1} \wedge b'_{n-1} \ldots b_0 \wedge b'_0]_s}\ \text{band}$$

For brevity we do not show other translation rules; the interested reader is referred to [18] for details.

A *condition* is a single bit, and includes the boolean constants, usual boolean operations, standard comparisons between expressions, and the `lift` of an expression, which is the disjunction of all the expression's bits. Similar to expressions, judgments of the form

$$[\![c]\!]_\psi = b$$

translate conditionals to boolean formulas.

A *statement* can be `skip` (a no-op), assignment, or `assert` or `assume` statements, which we explain shortly. Judgments

Language

Type (τ) ::= $(n, s) \mid \{(f_1, \tau_1), \ldots, (f_n, \tau_n)\}$ where $s \in \{\text{signed}, \text{unsigned}\}$

Obj (o) ::= $v \mid \{(f_1, o_1), \ldots, (f_n, o_n)\}$

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

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

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

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

Shorthand

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

Representation

Rep (β) ::= $[b_{n-1} \ldots b_0]_s \mid \{(f_1, \beta_1), \ldots, (f_n, \beta_n)\}$ where $s \in \{\text{signed}, \text{unsigned}\}$
Bit (b) ::= $0 \mid 1 \mid x \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid \neg b$

Figure 1: SATURN base language.

for statements have the form

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

where $\mathcal{G}$ and $\mathcal{G}'$ are *guards*. Guards are boolean formulas expressing the path-sensitive condition under which a statement actually executes. Consider the following example:

```
stmt_0;              /* G : true */
if (x > 5) stmt_1;   /* G : x > 5 */
else stmt_2;         /* G : ¬(x > 5) */
stmt_3;              /* G : (x > 5) ∨ ¬(x > 5) ≡ true */
```

We start with the guard true for stmt_0. The true branch of the first if statement is reached when $x > 5$ is true, so the guard for stmt_1 is $x > 5$. Similarly, the guard for the false branch is the negation of the branch predicate. stmt_3 is reached from both the true and false branches, so its guard is the disjunction of the predecessor guards. We use a BDD based algorithm to simplify guards before converting them to boolean formulas to avoid overwhelming the SAT solver with clauses in long chains of conditionals.

Statements are both guard and environment transformers: both the guard and the environment may be affected by the execution of a statement. A statement $\texttt{assume}(c)$ affects the guard; if $\mathcal{G}$ is the initial guard and b is the bit representing condition c, then the guard after the statement is $\mathcal{G} \wedge b$. Assignments transform the environment (by adding a new definition of the updated location for the next program point) and, in the presence of pointers, the guard (e.g., because on statement exit any indirect accesses in the statement must have succeeded and are known to be non-null).

An `assert`(c) statement causes SATURN to check that c is true at this program point.[2]

We define function bodies informally as control-flow graphs where the nodes are statements and directed edges are unconditional transfers of control. If a statement has multiple successors, we require that they be `assume` statements with disjoint conditions. A conditional branch with condition c is then modeled as a node with two successors, one of which is `assume`(c) and the other `assume`($\neg c$).

We must also define what happens when a node has multiple predecessors. We combine the various environments and guards of each of the predecessors into a single environment and guard, while preserving path sensitivity. The merged guard is just the disjunction of the guards of the predecessors, but the merged environment requires computing, for each bit b of each variable in the environment, the disjunction of the corresponding bits in each predecessor environment. Furthermore, each bit from each predecessor statement s is conditionally switched on by the guard from s; this preserves path sensitivity in the new environment. In the definitions below, we use the notation $\overline{X_i}$ as a shorthand for a vector of similar entities: $X_1 \ldots X_n$.

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

We can now describe at a high level how SATURN analyzes a function body. Beginning with the entry node of the control flow graph, SATURN simply traverses the graph in topological order, computing the guard and environment for each statement and issuing satisfiability queries at `assert` statements. A remaining issue is the handling of loops, because this simple procedure is not well-defined for back edges where the predecessor guard and environment have not yet been computed. For our leak checker we found it necessary to improve SATURN's loop analysis (see Section 3.5).

Finally, we outline how SATURN performs interprocedural analysis. The function call graph is analyzed bottom-up (currently recursion is broken arbitrarily and function pointers are modeled conservatively). Scalability is achieved by using function summaries, which are concise representations of functions' memory behavior. We use the inferred function summary in analyzing the callers of the function.

3. EXTENSIONS

In this section, we extend SATURN to support leak detection. We describe a path-sensitive pointer analysis that builds on *Guarded Location Sets (GLS)* [18] (Section 3.1); we extend GLS with type casts and polymorphic locations (Sections 3.2 and 3.3). We also introduce *attributes* (Section 3.4), which are annotations on objects and polymorphic locations. We discuss analyzing loops in Section 3.5.

3.1 Guarded Location Sets

Pointers in SATURN are modeled with *Guarded Location Sets (GLS)*. A GLS represents the set of locations a pointer could reference 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 3.3.

```
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).

3.2 Polymorphic Objects

The GLS representation models pointers to concrete objects with a single known type. However, in the leak analysis we often need to understand type casts to faithfully track memory locations. For example, in the following code,

```
void *malloc(int size);
p = (int *)malloc(len);
q = (char *) p;
return q;
```

the memory object allocated on the second line goes through three different types. These types all have different representations (i.e., different numbers of bits) and so must be maintained 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 the $\mathsf{AddrOf} : \mathsf{Obj} \rightarrow \mathsf{Addr}$ mapping 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. Using guarded location sets and addresses, we can describe the translation of pointers in detail.

3.3 Translation of Pointer Operations

Figure 2 defines the language and translation 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 translation rules work as follows. Taking the address of an object (get-addr-{obj,mem}) constructs a GLS with a

[2] If b is the translation of c and $\mathcal{G}$ is the guard of the statement, the assertion holds if $\mathcal{G} \wedge \neg b$ is unsatisfiable.

Language

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

Address

$$\mathsf{Addr}\ (\sigma) ::= \hat{1} \mid \hat{2} \mid \ldots$$
$$\mathsf{AddrOf} : \mathsf{Obj} \mapsto \mathsf{Addr}$$

(Constraint: no two objects of the same type share the same address)

Representation

$$\mathsf{Loc}\ (l) ::= \mathsf{null} \mid o \mid \sigma$$
$$\mathsf{Rep}\ (\beta) ::= \{\!|\ (\mathcal{G}_0, l_0), \ldots, (\mathcal{G}_k, l_k)\ |\!\} \mid \ldots$$

Translation

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

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

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

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

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

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

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

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

Figure 2: Pointers and guarded location sets.

Merging

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

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

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

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

Figure 3: Control-flow merges with pointers.

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 directly model explicit deallocation. In our leak checker, we treat deallocated memory blocks as escaped (Section 4.2). 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[3]. 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 and we omit the rule due to space limitations.

3.4 Attributes

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

The definition and translation 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 read and changed indirectly through pointers.

We omit the formal definition and translation 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 it is attached to addresses and preserved through pointer casts—thus the assertion at line 3 holds.

3.5 Loops

For our 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

[3]Recall $\mathcal{G}_0$ is the guard of null; see Section 3.1.

discarded; thus, just the first three iterations are analyzed. For leak detection this strategy works extremely 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. Intuitively, the second pass analyzes the last iteration of the loop; we model the fact that we do not know the state of the variables after an arbitrary number of earlier iterations by assigning them `unknown` values [19]. 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.

4. THE LEAK DETECTOR

In this section, we present a static leak detector based on the path sensitive pointer analysis described in Section 3. 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.

The rest of the section is organized as follows: Section 4.1 gives examples illustrating the targeted class of bugs and the analysis techniques required. We briefly outline the detection algorithm in Section 4.2 and give details in Sections 4.3, 4.4, and 4.5. Handling the unsafe features of C is described in Section 4.6. In Section 4.7, we describe a parallel client/server architecture that dramatically improves analysis speed.

4.1 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 path-sensitive as well:

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

To avoid false warnings in their path insensitive leak detector, Heine *et. al.* [8] transform this code into:

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

The transformation handles this 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 understand 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, some interprocedural analysis is required. Without abstraction, interprocedural program analysis is prohibitively expensive for path sensitive analyses such as ours. We adopt a summary-based approach that exploits the natural abstraction boundary at function calls. For each function, we use boolean satisfiability 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 4.5.

4.2 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 4.3) to track the path through which an object is accessed from both inside and outside (if possible) the function body. We describe details about how we model object accessibility in Section 4.4.

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

1. memory blocks may be freed;
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.,

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

Figure 4: Access paths.

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.

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 4.1 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.

4.3 Access Paths and Origins

This subsection describes how we track and manipulate the path through which objects are first accessed. As shown in the Section 4.2, path information is important in defining the escape condition for memory locations.

Figure 4 defines the representation and operations on *access paths*. 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. 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 1 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 4.4.

4.4 Escape and Leak Conditions

Figure 5 defines the rules we use to find memory leaks and construct function summaries. Without loss of generality,

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 1: Objects, access paths, and access origins in the sample program.

we make the following two assumptions about the control flow graph of the target function:

1. We assume that there is one unique exit block in each function's control flow graph. We handle multiple return statements in C by introducing a dummy exit-block linked to all return sites. The exit block is analyzed last (recall the topological order in which we process the control-flow graph blocks), and the environment ψ at the return site encodes all constraints for paths from function entry to exit. 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.

2. We model `return` statements in C by assigning the return value to the special object rv, whose access path is ret_val (recall Section 4.3).

In Figure 5, 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 PointsTo function, we are ready to define the escape relationships Escaped and EscapeVia.

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 5. 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 next rule 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

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

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

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

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

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

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

$$\mathsf{Escaped}(l,\mathcal{X}) = [\![l\#\mathsf{escaped}]\!]_\psi \vee \bigvee_p \mathsf{EscapeVia}(l,p,\mathcal{X}) \quad \text{escaped}$$

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

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

Figure 5: Memory leak detection rules.

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 4.1 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.

4.5 Interprocedural Analysis

This subsection describes the summary-based approach to interprocedural leak detection in SATURN. We start by defining the summary representation in Section 4.5.1, and discuss summary generation and application in Sections 4.5.2 and 4.5.3.

4.5.1 Summary Representation

Figure 6 shows the representation of a function summary. In leak analysis we are interested in whether the function returns newly allocated memory (i.e. allocator functions), and whether it creates any external reference to objects passed via parameters (recall Section 4.1). 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 4.3) to name escaped objects. These paths, called Escapees in Figure 6, are a subset of AccPath where the origin is a parameter.

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

Figure 6: The definition of function summaries.

IsMalloc:

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

Escapees:

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

Figure 7: Summary generation.

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

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

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 allocation. 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.

4.5.2 Summary Generation

Figure 7 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 (rv) 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 object rv 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 Escaped(l, {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.)

4.5.3 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 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 a valid SATURN object and must be transformed into one using a series of assignments. The details are omitted due to space limitations.

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 failure, 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 rv ← unknown.

4.6 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 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.

4.7 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 previously implemented in SATURN [18], the higher number of tracked objects (and thus SAT queries) means the leak analysis is much more computationally intensive.

However, the leak algorithm 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.

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. 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.

5. EXPERIMENTAL RESULTS

We have implemented the leak checker as a plug-in to the SATURN analysis framework [18] and applied it to five user space applications and the Linux kernel.

5.1 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 [8, 6]. 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

	LOC	Time	LOC/s	P.Time	P.LOC/s	Fn	Failed (%)	Alloc	Bugs	FP (%)
Samba	403,744	3h22m52s	33	10m57s	615	7,432	24 (0.3%)	80	83	8 (8.79%)
OpenSSL	296,192	3h33m41s	23	11m09s	443	4,181	60 (1.4%)	101	117	1 (0.85%)
Postfix	137,091	1h22m04s	28	12m00s	190	1,589	11 (0.7%)	96	8	0 (0%)
Binutils	909,476	4h00m11s	63	16m37s	912	2,982	36 (1.2%)	91	136	5 (3.55%)
OpenSSH	36,676	27m34s	22	6m00s	102	607	5 (0.8%)	19	29	0 (0%)
Total	1,783,179	12h46m22s	39	56m43s	524	16,791	136 (0.8%)	387	373	14 (3.62%)

(a) User space applications.

	LOC	Time	LOC/s	P.Time	P.LOC/s	Fn	Failed (%)	Alloc	Bugs	FP (%)
Linux v2.6.10	5,039,296	23h13m27s	60	50m34s	1661	74,367	792 (1.06%)	368	82	41 (33%)

(b) Linux Kernel 2.6.10.

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

Table 2: Experimental Results.

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

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

```
1 /* Samba – client/clitar.c:do_tarput */
2 longfilename = get_longfilename(finfo);
3 ...
4 return;
```

(b) get_longfilename allocates new memory.

```
1 /* Samba – utils/net_rpc.c:rpc_trustdom_revoke */
2 domain_name = smb_xstrdup(argv[0]);
3 ...
4 if (!trusted_domain_password_delete(domain_name))
5     return −1;
6 return 0;
```

(c) trusted_domain_password_delete does not free.

Figure 8: Three bugs found by the leak checker.

80 idle workstations[4]. For each function, the resource limits are set to 512MB of memory and 90 seconds of CPU time.

Table 2(a) gives the performance statistics and bug counts. 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 387 warnings across these applications. We have examined all the warnings and believe 373 of them are bugs. (Warnings are per allocation site to facilitate inspection.) Besides bug reports, the leak checker generates

[4]We constantly monitor CPU load and user activity on these machines, and avoid using clients that have active users or tasks.

```
1 /* OpenSSL – crypto/bn/bn_lib.c:BN_copy */
2 t = BN_new();
3 if (t == NULL) return (NULL);
4 r = BN_copy(t, a);
5 if (r == NULL)
6     BN_free(t);
7 return r;
```

Figure 9: A sample false positive.

a database of function summaries documenting each function's memory behavior. In our experience, 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.

Figure 8 shows representative errors we found. The errors largely fall into three categories: missed deallocation on error paths (Figure 8a), covert allocators (Figure 8b), and misunderstood function interfaces (Figure 8c). Figure 9 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, and this is not detected, nor is it expressible by the function summary.

5.2 The Linux Kernel

Table 2(b) summarizes statistics of our experiments on Linux 2.6.10. Using the parallel analysis framework (recall Section 4.7) we distributed the analysis workload on 80 CPUs. Our implementation monitors system activity and takes clients offline when they are under load or when there are active users. 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 are due to pointer arithmetic. Application specific logic accounted for another 12, and the remaining 4 are due to SATURN limitations in modeling constructs such as arrays and unions.

6. RELATED WORK

Previously we used SATURN to implement a path- and context-sensitive analysis for detecting a class of locking discipline errors. This analysis finds many more locking errors with lower false positive rates than previously published studies and the analysis also scales to millions of lines of code [18]. Detecting memory leaks is a very different and more complex property than analyzing more purely finite state properties such as locking. In particular, a much more sophisticated analysis of pointers and the heap is required.

Jackson and Vaziri were apparently the first to consider finding bugs via reducing program source to boolean formulas [10]. Subsequently there has been significant work on a similar approach called *bounded model checking* [11]. 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 checking of properties without separate programmer-written specifications.

Memory leak detection using dynamic techniques 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* [7]; see [2] 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 [5], 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 [1] 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 [3] 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) [8]. 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) [6]. 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 to be likely 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 [12] as symbolic names for memory locations accessed in a procedure. Several later algorithms (e.g., [4, 17, 13]) also make use of parameterized pointer information to achieve context sensitivity. Escape analysis (e.g. [16, 14]) 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.

Other systems have investigated encoding C pointers using boolean formulas. CBMC [11] uses uninterpreted functions. SpC [15] uses static points-to sets derived from an alias analysis. There, the problem is much simplified since the points-to relationship is concretized at execution time and integer tags (instead of boolean formulas) can be used to guard the points-to relationships. F-Soft [9] models pointers by introducing extra equivalence constraints for all objects reachable from a pointer, which is inefficient in the presence of frequent pointer assignments.

7. CONCLUSION

We have presented a novel memory leak detection algorithm based on solving boolean satisfiability constraints. Scalability is achieved by querying boolean formulas representing each function to produce a concise function summary. Experimental results show that our system scales well, parallelizes well, and finds more leaks with many fewer false positives than previous leak detection systems.

Acknowledgements

We would like to thank the Stanford database group for generously donating unused CPU cycles for the distributed checking experiment; Intel Corporation for donating Pentium 4 workstations and a dual processor Xeon server on which this research is conducted. We are also grateful to Ted Kremenek, Mayur Naik, Tachio Terauchi, Junfeng Yang, and the anonymous referees for their helpful comments and discussions on an earlier draft of the paper.

8. REFERENCES

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

[2] T. Chilimbi and M. Hauswirth. 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*, 2004.

[3] A. Chou. *Static Analysis for Bug Finding in Systems Software*. PhD thesis, Stanford University, 2003.

[4] M. Emami, R. Ghiya, and L. Hendren. 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*, 1994.

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

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

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

[8] D. L. Heine and M. S. Lam. 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*, pages 168–181, 2003.

[9] F. Ivancic, Z. Yang, M. Ganai, A. Gupta, and P. Ashar. Efficient SAT-based bounded model checking for software verification. In *Proceedings of the 1st International Symposium on Leveraging Applications of Formal Methods*, 2004.

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

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

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

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

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

[15] L. Semeria and G. D. Micheli. SpC: synthesis of pointers in C: application of pointer analysis to the behavioral synthesis from C. In *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 340–346. ACM Press, 1998.

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

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

[18] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *Proceedings of the 32nd Annual Symposium on Principles of Programming Languages*, Jan. 2005.

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