How is Aliasing Used in Systems Software?

Brian Hackett Alex Aiken
Computer Science Department
Stanford University

Abstract

We present a study of all sources of aliasing in over one million lines of C code, identifying in the process the common patterns of aliasing that arise in practice. We find that aliasing has a great deal of structure in real programs and that just nine programming idioms account for nearly all aliasing in our study. Our study requires an automatic alias analysis that both scales to large systems and has a low false positive rate. To this end, we also present a new context-, flow-, and partially path-sensitive alias analysis that, together with a new technique for object naming, achieves a false aliasing rate of 26.2% on our benchmarks.

General Terms

Verification, Experimentation

Categories and Subject Descriptors

Software [**Software Engineering**]: Program Verification

Keywords

aliasing, program analysis, satisfiability

1. INTRODUCTION

Many program analyses have been designed to model aliasing relationships and many methods have been developed for the control and specification of aliasing. However, to the best of our knowledge, there has been no systematic attempt to document or describe how and how often aliasing is actually used in large systems. We set out to try.

Our long-term goal is to develop an alias analysis sufficiently precise and scalable to be used in sound verification tools for multi-million line software systems, and as a first step we wanted to carry out a study of the important aliasing patterns and their frequency in real code. Unfortunately, current alias analyses that scale to even one million lines of code are known to be imprecise with respect to the aliasing that actually exists at run-time [21, 25]. More precise approaches using *context-sensitive* or *object-sensitive* [23] analysis have recently been shown to scale to hundreds of thousands of lines [3, 29, 26] while *flow-* or *path*-sensitive approaches have scaled to tens of thousands of lines [7, 31, 22]. However, we needed all these features to understand the aliasing structure of large programs without being overwhelmed by false positives, as well as to support the larger goal of verification.

This paper makes three primary contributions. First, we present a new *context-*, *flow-*, and partially *path*-sensitive alias analysis, as well as a new, precise form of *object naming* that helps greatly in keeping information about distinct data structures distinct. (For the reader unfamiliar with the technical jargon of program analysis, we explain these terms in Section 3.) Path-sensitivity is only intraprocedural; the other features are not restricted, and in particular context-sensitivity and object naming are not k-limited [16]. Our analysis is sound and fully automatic, although it can also take advantage of user-supplied alias annotations.

Second, we show this analysis scales well; our largest benchmark is nearly 600,000 LOC. We have used the analysis on programs with millions of lines of code, though we do not include such programs in this paper—the manual effort required to classify all aliasing in multi-million line programs is beyond our resources.

Third, we use the analysis to carry out a study of how aliasing is used in large C system programs and in the process show that the analysis is much more precise than previously reported results for static alias analysis of large systems. Using our analysis, we have identified and classified by hand all sources of aliasing in over one million lines of C code. The *false aliasing rate* (the fraction of discovered aliasing relationships that are incorrect due to analysis imprecision) is 26.2%, which is much lower than the false aliasing reported by previous efforts to measure the precision of static alias analysis [25, 21]. This accuracy is also what makes the system usable for our study.

We have identified and measured the frequency of the idioms programmers use to create aliasing (Section 5). Perhaps surprisingly, we have found that there is little diversity in aliasing: just nine patterns account for almost all the aliasing in our study. We believe this data will be useful in the design of software tools and language constructs that deal with aliasing. In fact, we used early results of this study to help design our alias analysis for scalability. We also find some examples of apparently unintentional and potentially dangerous aliasing; one such instance has been confirmed

SIGSOFT'06/FSE-14, November 5–11, 2006, Portland, Oregon, USA.

as a subtle bug in PostgreSQL, a widely used open source database system (Section 5.1).

Pointer aliasing occurs when multiple pointers are used to access the same data. Any pointer copy creates the potential for aliasing, if both the original and copy are later used. A key insight that we gained from our study is that in real programs aliasing occurs on multiple different scales reflecting the structure of the program. For example, there is aliasing specific to particular data types, and in fact all instances of a type often have a specific aliasing relationship throughout the program; in our study, 7.8% of user-defined types (`struct`'s in C) are associated with aliasing. Other examples of structure in aliasing is aliasing associated with particular global pointers or particular functions.

We have used the fact that aliasing occurs at different scales in the design of our analysis. Consider a tree data structure with parent pointers. It is wasteful and expensive to rediscover the aliasing of the parent through the child at every use of that data type in a large program. However, by treating this aliasing as a program-wide *aliasing invariant*, we can record the information in one place instead of at (potentially) every program point. Factoring out aliasing specific to specific types and global variables dramatically reduces the amount of data that must be represented. Our results show that outside of the user-defined data structures that use aliasing, aliasing is actually rare. In our study, only 5.1% of all global pointers are involved in aliasing. If one discounts aliased data types and global pointers, only 3.5% of functions and 13.2% of loops make use of additional aliased values. Because aliasing outside of globals and data types is so rare, we can use precise methods to track it with reasonable cost.

1.1 Threats to Validity

There are four threats to the validity of our study. The first is that the systems programs in our study may not be representative of programs in other domains. For example, object-oriented programs may have more aliasing patterns than the ones we discovered.

Second, our alias analysis can be used either automatically or with hand annotations. We believe fully automatic alias analysis is often undesirable in verification of low-level C programs; a very few annotations can capture user-knowledge that is difficult to infer automatically. For this study we used 19 annotations on custom memory allocators to treat them like `malloc`. (We did not annotate `malloc` wrappers, which we infer automatically.) As far as we know, all alias analyses assume primitive allocation routines are specified as part of the input.

Third, the manual process of inspecting the alias analysis results by hand introduces a source of error; we may have made mistakes in categorizing the sources of aliasing discovered by the analysis. We believe the misclassification rate is quite low, because we have found the vast majority of aliasing is simple and easy to classify. Also, over the course of this study we did the full manual analysis of all the benchmarks several times as we refined our methodology, giving multiple opportunities for mistakes to be found and corrected.

Fourth, we believe our alias analysis is sound for C under a few common assumptions (e.g., assuming memory safety and that programs do not hide or manufacture pointers). However, we have not done a formal proof of soundness.

```
struct packet {
  struct packet *next;
  struct packet **prev;
  u_char type;
  Buffer payload;
};
struct packetlist {
  struct packet *first;
  struct packet **last;
} out;

void main(...) {
  packet_set_conn(...);
  while (...) {
    packet_send2(); }
}
```

```
void packet_set_conn(...) {
    if (...) {
    out.first = NULL;
    out.last = &out.first; }
}

void packet_send2() {
  struct packet *p;
  if (...) {
    p = xmalloc(sizeof(packet));
    p->prev = out.last;
    *out.last = p;
    out.last = &(p->next); }
  if (...) {
    p = out.first; }
}
```

Figure 1: An example program.

2. ALIASING STRUCTURE

Real programs use aliasing at multiple levels of granularity. Compactly and precisely representing aliasing requires exploiting this structure. We have identified four commonly used levels of aliasing structure:

- *entry aliasing* specific to the entry state of a function.
- *exit aliasing* specific to the exit state of a function.
- *global invariant aliasing* specific to a global variable.
- *type invariant aliasing* specific to all values of a type.

The following example illustrates these levels.

EXAMPLE 1. Code adapted from `openssh-3.9p1/packet.c` is in Figure 1. A doubly linked list of `packets` is managed using head and tail pointers in global structure `out`. The `last` and `prev` pointers hold addresses of either the `first` or `next` pointers in the list. Function `packet_set_conn` is called before `packet_send2`, which may be called any number of times. Function `packet_set_conn` introduces aliasing between `*out.last` and `out.first`, affecting later calls to `packet_send2`.

We could interpret this aliasing at multiple levels:

- This aliasing is particular to the exit state of `packet_set_conn` and entry state of `packet_send2`.
- This aliasing is a property of global variable `out`, where `out.last` may generally point to `out.first`.
- This aliasing is a property of all values of type `packetlist`, where `last` may generally point to `first`.

These interpretations are all technically correct, but differ in how well they describe the program's behavior. The last best fits our intuitive understanding of the program's structure: this aliasing is a property of the `packetlist` type. Under the first two interpretations, we must redundantly account for similar aliasing on other variables of type `packetlist`, or other functions manipulating such variables. □

2.1 Documenting Aliasing

In this section, we describe how the levels of aliasing structure are documented in our algorithm, and how this documentation is combined to produce aliasing information for the entire program. We assume a conventional typed imperative language with functions, dynamic allocation of typed data, pointers, and field accesses and updates; we specify such a language more precisely in Section 3.

To discuss aliasing we need to name memory locations so that we can say when two pointers point to the same location. We represent locations by *labels*, which are sequences of field accesses and pointer dereferences from a variable name or the special name this, which we discuss shortly:

$$l \in \text{Label} \quad ::= \quad x \mid \mathsf{this} \mid l.f \mid *\, l$$

Labels may be either *singleton*, representing at most one location at any point, or *unbounded*, representing arbitrarily many locations. Unbounded labels may be arrays or sets of elements of recursive structures. The predicate $\mathsf{unbounded} : \text{Label} \to \{\mathsf{true}, \mathsf{false}\}$ distinguishes unbounded labels from singletons.

We represent aliasing indirectly using *guarded points-to graphs*. In a *points-to graph* nodes are labels and edges (l_1, l_2) mean that pointers at locations in l_1 may point to locations in l_2. Aliasing information is recovered easily from points-to graphs; for example, edges (x, l) and (y, l) show that x and y may be aliased, as they both may point to locations in label l. Guarded points-to graphs generalize points-to graphs by associating each edge with a *guard*, a predicate stating under what condition the points-to relationship holds. Guards contribute to the precision of our analysis; for example, if x and y may both point to l, but with guards that cannot simultaneously hold, they are not aliased. We use formulas over boolean variables b for guards.

$$\begin{aligned} g \in \text{Guard} \quad &::= \quad \mathsf{true} \mid \mathsf{false} \mid b \mid g_0 \wedge g_1 \mid g_0 \vee g_1 \mid \neg g \\ \rho \in \text{PTGraph} \quad &= \quad (\text{Label} \times \text{Label}) \to \text{Guard}_\perp \end{aligned}$$

Points-to graphs are *partial*—they may not be defined on all edges. An undefined edge is represented by $\text{PTGraph}(l_1, l_2) = \perp$. Partial graphs allow us to distribute the full set of edges across several graphs.

Our algorithm is *summary-based*: for each function f we summarize f's aliasing behavior and use only the summary in analyzing sites where f is called. A summary for f is a pair (ρ_{in}, ρ_{out}) of partial points-to graphs, ρ_{in} for entry aliasing and ρ_{out} for exit aliasing of f.

The summary points-to graphs are partial, defined only for labels involved with entry or exit aliasing. *Targets* (i.e., pointed-to labels) for all other labels are given by invariants on global variables and user-defined types. Partitioning points-to information into summaries encoding what is unique to each function and a set of global invariants is a key aspect of our approach.

The points-to graph ρ_{inv} records the invariant targets of each label. Any partial points-to graph ρ lifts to a *complete* points-to graph $\bar{\rho}$ defined for all edges, which uses ρ_{inv} to get the targets of labels not in ρ:

$$\bar{\rho}(l_1, l_2) \quad = \quad \textbf{if } \rho(l_1, l_2) \neq \perp \textbf{ then } \rho(l_1, l_2) \textbf{ else } \rho_{inv}(l_1, l_2)$$

We factor ρ_{inv} into ρ_g, ρ_τ, and ρ_{def}, representing, respectively, invariants on global variables and each type τ (a separate graph for each type), and default points-to relationships for edges not otherwise defined.

$$\rho_{inv}(l_1, l_2) = \begin{cases} \rho_g(l_1, l_2) & \text{if defined} \\ \rho_\tau(l_1, l_2) & \text{if defined for } \tau \\ \rho_{def}(l_1, l_2) & \text{otherwise} \end{cases}$$

The default points-to graph ρ_{def} sets the only target of each label l to $*\, l$. Since this value is unique to l, no two labels have aliased targets by default.

$$\forall l_1, l_2. \quad \rho_{def}(l_1, l_2) \Leftrightarrow (l_2 = *\, l_1)$$

Type invariants use the special label this to quantify over all labels of the type. For example, $(\mathsf{this}.f, *\mathsf{this}.g)$ indicates for all values of type t, fields f and g are aliased. The this notation compactly represents type invariants, but requires instantiation of the this to perform lookups on ρ_τ. For simplicity we write $\rho_\tau(l_1, l_2)$, treating ρ_τ as a map where all instances are already instantiated.

We return to the program from Example 1, showing the different ways that the observed aliasing can be documented.

EXAMPLE 2. Consider again the aliasing between `*out.last` and `out.first` that may exist in Figure 1. This aliasing can be documented at the function, global, or type level.

To document aliasing at the function level, the exit aliasing on `packet_set_conn` and entry aliasing on `packet_send2` are added to these functions' summaries. In particular, the points-to edge (`out.last`, `out.first`) with guard true is added to ρ_{out} of `packet_set_conn` and to ρ_{in} of `packet_send2`. This aliasing must also be documented for the many other functions which directly or indirectly access `out`.

At the global level, `out.last` always points to the last next pointer in the list, which is either the first pointer or a next pointer. We add the global invariants

$$(\texttt{out.last}, \texttt{out.first}), \quad (\texttt{out.last}, \texttt{*(out.first).next})$$

Any other values of type `packetlist` must also be documented. At the type level, the invariants

$$(\texttt{this.last}, \texttt{this.first}), \quad (\texttt{this.last}, \texttt{*(this.first).next})$$

document aliasing for all values of type `packetlist`. □

3. CHECKING AND INFERENCE

In this section, we first give an algorithm to check the correctness of given points-to information. We discuss flow- and path-sensitive intraprocedural analysis (Section 3.1), context-sensitive interprocedural analysis of function calls (Section 3.2), and finally our object naming scheme (Section 3.3). Section 3.4 extends checking to an algorithm for inferring function summaries.

Consider a function f that calls function g; we want to check that given summaries for f and g are correct. Assume the entry state for f satisfies its entry points-to graph, and that at the call to g the exit state satisfies g's exit points-to graph. To verify correctness of the points-to information we must check two things:

1. The exit state for f satisfies f's exit points-to graph.
2. The entry state for the call to g satisfies g's entry points-to graph.

If conditions (1) and (2) are satisfied simultaneously for all functions and call sites, then the points-to information is consistent and correct.

We present our analysis using a simple imperative language (but we use examples that go outside this language). A program consists of global variables, types, and functions. A function body is a *command* and a set of local stack variables; arguments are passed to functions through assignment to global variables. Basic commands c are stores, calls, and dynamic allocations, and are composed with sequences and

non-deterministic branches. Functions are loop-free; in our implementation for C, loops are represented as tail-recursive functions for which summaries are required. Expressions e are either the address of a variable, the sum of another expression and the fixed offset of a field, or the dereference of another expression.

$$\begin{array}{rcl} e & ::= & \&x \mid e + f \mid *e \\ c & ::= & c_0; c_1 \mid \textbf{if } ?\ c_0\ c_1 \mid e_0 \leftarrow e_1 \mid \textbf{call } f \mid e \leftarrow \textbf{new}(\tau) \end{array}$$

A points-to graph ρ_2 is more conservative than ρ_1 if whenever (l_1, l_2) holds in ρ_1 it also holds in ρ_2. For a set of edges L, we define a relation $\leq_L$ on points-to graphs:

$$\rho_1 \leq_L \rho_2 \Leftrightarrow \forall (l_1, l_2) \in L.\ \rho_1(l_1, l_2) \Rightarrow \rho_2(l_1, l_2)$$

where $\perp$ (i.e., an undefined edge) is interpreted as the predicate `false`. To check correctness of a function f with body c against an aliasing summary (ρ_{in}, ρ_{out}), we apply the *transfer function* (defined below) $[\![c]\!]$ to ρ_{in} to compute the exit points-to graph ρ and verify that

$$\rho \leq_L \rho_{out} \text{ where } L = \mathsf{dom}(\rho_{out}) \cup \mathsf{dom}(\rho) \quad (*)$$

where $\mathsf{dom}(f)$ is the domain (the edges) on which points-to graph f is defined. (We also sometimes abuse notation and write $l \in \mathsf{dom}(f)$, treating the set of edges as the set of all nodes mentioned in the edges.)

3.1 Function Body Evaluation

The following C fragment conditionally swaps two values:

```
a = x; b = y; // *x != *y
if (...) { t = a;  a = b;  b = t; }
f(a,b);
```

It is easy to see that `a` and `b` are never aliased at the call `f(a,b)`. To prove this fact, the analysis must be *flow-sensitive*, that is, it must respect the order of assignments in the branch. (In contrast, *flow-insensitive* analysis ignores the order of statements and thus loses information about the order of side effects.) We also need *path-sensitivity*, meaning we must distinguish the two execution paths to recognize that `a` and `b` never point to the same value on the same path.

A *transfer function* $[\![c]\!]$ models the effect of command c on a partial points-to graph ρ (possibly also using the global invariants ρ_{inv} as needed). Thus, a transfer function maps points-to graphs to points-to graphs. The transfer function for command sequences is standard:

$$[\![c_0; c_1]\!]\rho \quad = \quad [\![c_1]\!]([\![c_0]\!]\rho)$$

We write function application $[\![c]\!](\rho)$ using juxtaposition $[\![c]\!]\ \rho$ to avoid cluttering definitions with parentheses.

For path-sensitivity, fresh boolean variables are used to separate points-to information along branches. Even without branch condition information (i.e., ignoring the actual predicate of the conditional), path information can track correlations between side effects, as in the swap example above.[1]

$$\begin{array}{l} [\![\textbf{if } ?\ c_0\ c_1]\!]\ \rho\ (l, l') \ = \\ \quad \textbf{let } \rho_0 = [\![c_0]\!]\rho \textbf{ in} \\ \quad \textbf{let } \rho_1 = [\![c_1]\!]\rho \textbf{ in} \\ \quad \textbf{let } b \text{ be a fresh boolean variable } \textbf{in} \\ \qquad (b \wedge \bar{\rho}_0(l, l')) \vee (\neg b \wedge \bar{\rho}_1(l, l')) \end{array}$$

[1]Our implementation for C also analyzes branch conditions as in [32].

The transfer function for store commands uses $\mathcal{E}[\![e]\!]$, which maps a pointer expression e to a function from e's labels to their guards under a given points-to graph:

$$\begin{array}{rcl} \mathcal{E}[\![e]\!]\rho & \in & \text{Label} \rightarrow \text{Guard} \\ \mathcal{E}[\![\&x]\!]\rho\ l & = & \textbf{if } l = x \textbf{ then } \mathsf{true} \textbf{ else } \mathsf{false} \\ \mathcal{E}[\![e + f]\!]\rho\ l & = & \textbf{if } l = l'.f \textbf{ then } \mathcal{E}[\![e]\!]\rho\, l' \textbf{ else } \mathsf{false} \\ \mathcal{E}[\![*e]\!]\rho\ l & = & \bigvee_{l'} (\mathcal{E}[\![e]\!]\rho\, l' \wedge \bar{\rho}(l', l)) \end{array}$$

The transfer function for stores adds the potential targets of the source to the potential targets of the destination. If a destination target is singleton, its old targets are removed; this *strong update* is made possible by flow-sensitivity and singleton labels [5].

$$[\![e_0 \leftarrow e_1]\!]\rho \ = \ \mathsf{store}(\rho, \mathcal{E}[\![e_0]\!]\rho, \mathcal{E}[\![e_1]\!]\rho)$$

$$\begin{array}{l} \mathsf{store}(\rho, m_0, m_1)\ (l, l') \ = \\ \quad (m_0\, l \wedge m_1\, l') \ \vee \ (\bar{\rho}(l, l') \wedge (\mathsf{unbounded}(l) \vee \neg m_0\, l)) \end{array}$$

The auxiliary function store defines the points-to graph resulting from the store command. In this graph, l points to l' if l was assigned l' by the store command (the predicate $m_0\, l \wedge m_1\, l'$), or l pointed to l' in the old state and l is either unbounded or was not the target of the assignment (the predicate $\bar{\rho}(l, l') \wedge (\mathsf{unbounded}(l) \vee \neg m_0\, l)$).

Returning to the swap example, let `z` be the value of the condition and let `a`, `b`, and `t` be singleton (local variables are always singleton). At the call site we have $(\mathtt{a}, *\mathtt{y}), (\mathtt{b}, *\mathtt{x})$ under guard `z` and $(\mathtt{a}, *\mathtt{x}), (\mathtt{b}, *\mathtt{y})$ under guard $\neg$`z`.

3.2 Function Call Evaluation

Consider the following C code fragment:

```
int *h, *g, *x, *y;
int *f(int *a) { return a; }
x = f(h);  // 1
y = f(g);  // 2
```

To accurately analyze this example, we must separate the two calls to `f` so that we discover the points-to facts `(x,*h)` at call site (1) and `(y,*g)` at call site (2) but not, say, that `(x,*g)`. This is *context-sensitivity*, the ability to analyze a function separately in each context where it is called.

For alias analysis a key aspect of context-sensitivity is that a function's summary need not be concerned with labels the function does not access. In the example above, `f`'s summary need not mention global variables `h` and `g` or the aliasing relationship between those variables and `f`'s formal parameter `a`, because while the globals are in scope in `f` they are not used by `f`. From `f`'s point of view `a` is locally unaliased or *restricted* [1]: `f`'s sole access to `*a` is through the name `a`, and thus `f`'s behavior is polymorphic in `a`, regardless of what aliases exist outside of `f`. To exploit this form of polymorphism we require sets of accessed labels ψ for each function call site (i.e., the set of labels read or written by the call). While these sets are often considerably larger than the function summaries, they are easy to compute using a simple, safe over-approximation. Returning to the example, $\psi_1 = \{\mathtt{h}\}$ at call site (1) and $\psi_2 = \{\mathtt{g}\}$ at call site (2).

Consider a call site where ρ is the caller's points-to graph and ρ_{in} is the input points-to graph of the callee's summary. Recall that points-to graphs are partial, and that the domains of ρ and ρ_{in} are different (because the caller and

callee functions have different name spaces). Thus we need a *label renaming* mapping labels l in ρ to corresponding labels $\sigma(l)$ in ρ_{in}. We lift renamings to points-to graphs:

$$\sigma(\rho')(l_1, l_2) = \rho'(\sigma(l_1), \sigma(l_2))$$

The correctness condition relating the caller and callee is that the callee's input points-to graph must be conservative with respect to the caller's points-to graph for corresponding labels under the renaming:

$$\rho \leq_{\psi \cap \mathsf{dom}(\rho)} \sigma(\rho_{in})$$

where σ is the label renaming and

$$\psi \cap \mathsf{dom}(\rho) = \{(l_1, l_2) \in \mathsf{dom}(\rho) | l_1, l_2 \in \psi\}$$

The condition must hold only for labels accessed by the callee (ψ) and observed by the caller ($\mathsf{dom}(\rho)$); thus, there is polymorphism in what the caller does not observe and what the callee does not use.

The crux of the test given above is computing a renaming satisfying the condition; renaming fails when there are alias relationships in the caller not modeled in the callee. To compute σ for our restricted language initially $\sigma(x) = x$ for any globals shared between the caller and callee (for C, we also include renamings between formal and actual parameter names in the initial renaming). We then extend σ to satisfy the following two conditions for labels in $\psi \cap \mathsf{dom}(\rho)$:

$$\begin{array}{rcl} \sigma(l.f) & = & \sigma(l).f \\ \rho(l_1, l_2) & \Rightarrow & \rho_{in}(\sigma(l_1), \sigma(l_2)) \end{array}$$

The second condition says that if l_1 points to l_2, then $\sigma(l_2)$ must be chosen to be a member of $\sigma(l_1)$'s points-to set; any choice satisfying the implication is sound.

The call transfer function uses the label renaming to add new aliasing relationships resulting from the call:

$[\![\textbf{call } f]\!]\rho\ (l_1, l_2)\ =$
  let $(\rho_{in}, \rho_{out}) =$ *summary of* f **in**
  let $\psi =$ *labels accessed by* f **in**
  let $\sigma =$ *a satisfying label renaming on* $\psi \cap \mathsf{dom}(\rho)$ **in**
    $\bar{\rho}(l_1, l_2) \vee \bar{\rho}_{out}(\sigma(l_1), \sigma(l_2))$

Returning to this subsection's example, both the input and output points-to graph of `f` is `(a,*a)` (note globals `h` and `g` are not mentioned). At call site (1) the renaming $\sigma_1(\texttt{h}) = \texttt{a}$ allows us to prove that `x` points to `*h` after the call; at call site (2) the renaming $\sigma_2(\texttt{g}) = \texttt{a}$ shows that `y` points to `*g`.

3.3 Allocation Sites

Consider the following program fragment:

```
f(int **a) { *a = h(); ... }
g(int **b) { *b = h(); ... }
int *h()   { return (int *) malloc(...); }
```

It is important that the treatment of dynamically allocated data distinguish among data allocated at different call sites. Otherwise we may conflate reused structures such as generic lists and hash tables, as functions allocating such structures are also typically reused. In the example above, we wish to know that `a` and `b` are different, despite the fact that a single allocation site in `h` allocates both. This issue is known as *object naming* [23]. In our summary-based approach, the problem reduces to integrating freshly allocated data with a function's summary.

Allocation is normally used to grow a pre-existing structure or to replace a pre-existing value. For each allocated value, then, we find a *role label*, an existing label that represents the role the new value serves in the data manipulated by the function. As we cannot determine whether dynamically allocated locations escape and have a role until the allocating function exits, we give new locations a fresh label and rename them to their roles at exit.

The transfer function for an allocation statement creates a fresh label x and stores a pointer to it.

$$[\![e \leftarrow \textbf{new}(\tau)]\!]\rho \quad = \quad \mathsf{store}(\rho, \mathcal{E}[\![e]\!]\rho, \lambda l.(l = x))$$

At function exit we remove fresh labels from a points-to graph ρ by assigning roles consistent with an exit points-to graph ρ_{out}. We construct a label renaming π such that for each newly allocated label $x \in \mathsf{dom}(\rho)$ we have $\pi(x) \in \mathsf{dom}(\rho_{out})$. We need to find a π such that

$$\rho \leq_L \pi(\rho_{out}) \text{ where } \quad L = \mathsf{dom}(\rho_{out}) \cup \mathsf{dom}(\rho)$$

which refines the exit correctness condition ($*$) given at the beginning of Section 3.

We construct π as follows. For each newly allocated label x, we set $\pi(x) = l_2$ if there is a label l_1 such that $\rho(l_1, x)$ and $\rho_{out}(l_1, l_2)$. Intuitively, if l_1 points to x in ρ, then any non-new value l_1 points to in ρ_{out} is a good role for x. If there are multiple possibilities for l_2 we pick one arbitrarily. While this heuristic can fail in the sense that it may produce poor roles, in our experience that is rare because unique roles are extremely common.

Consider the function `h` in the example above. For C, we model return values as assignment to a special variable `return`. On exit, we have the points-to relation `(return,x)`, where `x` is the newly allocated location. Since `return` does not point to anything but its default `*return` in the exit summary, `x` is renamed to `*return`. An output points-to relation `(return,*return)` is the signature of a function returning freshly allocated memory (e.g., it is the signature of `malloc`) and we treat such functions analogously to **new** above, introducing a new, distinct label at each call site. These new labels will in turn be renamed according to the roles of `a` and `b` in functions `f` and `g`, respectively.

3.4 Summary Inference

The core of the checking algorithm is verifying $\rho \leq_L \rho'$ for two guarded points-to graphs ρ and ρ'. Testing $\leq_L$ requires computing for each $(l_1, l_2) \in L$ whether $\rho(l_1, l_2) \Rightarrow \rho'(l_1, l_2)$, which can be done using a solver for boolean satisfiability.

The checking algorithm assumes that global invariants and function summaries are supplied and verifies their correctness; we now extend the checking algorithm to an inference algorithm that also computes invariants and function summaries. We begin with empty summaries for all functions and an empty set of invariants and run the checking algorithm. When checking fails at function exit, callee aliasing is not reflected in the caller; when checking fails at a call site, caller aliasing is not reflected in the callee. We add the alias relationship(s) that caused checking to fail to the function summaries. The points-to relationships of function pointers are discovered as part of the analysis (just like for other pointers), allowing the call graph to be constructed on-the-fly. The process of checking function summaries and reanalyzing functions if checking fails repeats until checking succeeds simultaneously for all functions, which is clearly

			Invariants		Functions			Loops		
Application	KLOC	Time	$\mathbb{A}_{type}$	$\mathbb{A}_{glob}$	Count	$\mathbb{A}_{in}$	$\mathbb{A}_{out}$	Count	$\mathbb{A}_{in}$	$\mathbb{A}_{out}$
openssh-3.9p1	64	2:45	14	15	1147	43	145	492	72	203
openssl-0.9.7e	225	21:28	185	120	4109	180	1662	1381	209	963
httpd-2.0.53	230	12:27	149	91	1980	102	1373	1070	308	1049
postgresql-8.0.2	575	4:22:06	597	343	7842	810	2720	4262	1431	4626
Overall	1094	4:58:06	945	569	15078	1135	5900	7205	2020	6841

Table 1: Analysis results: application size, analysis time, type and global invariant aliasing points-to edges, and summary counts and instances of entry and exit aliasing generated for functions and loops.

sound. The remaining issues are termination and efficiency.

To guarantee termination, we use unguarded points-to graphs in function summaries (i.e., the guards in summaries are all either `true` or `false`; any satisfiable guard in a summary is promoted to `true`). This conservative approximation has two practical ramifications. First, it limits path sensitivity to within function bodies, as no non-trivial guards are propagated between functions. As a result, the size of the largest boolean formula that must be solved is limited by the size of the largest function, and because function size does not grow (or grow very much) with the size of a program, we are not limited by the scalability of SAT solvers. Second, termination becomes much easier to reason about if summaries do not include guards. If the set of labels (nodes) in an unguarded points-to graph is bounded, it follows that the size of the unguarded points-to graph is bounded. Since the inference procedure only monotonically adds unguarded points-to edges to summaries, inference must terminate.

To bound the set of labels it suffices to bound label length. We require that all types of locations in labels be distinct; since a program has a finite number of types, this bounds the length of the longest label. During inference a label may arise that would have the same type in two positions; in that case we identify the two positions together, which may require making some singleton values unbounded and combining points-to sets of the formerly distinct locations. This identification is just the standard approximation of collapsing recursive data types; for example, instead of trying to distinguish each element of list, the approximation treats all list elements as one, unbounded location.

Another issue is that **new** introduces new labels. However, role renaming guarantees the new labels do not appear in function summaries and so cannot affect termination.

Turning to efficiency, this approach often fails to halt in reasonable time even for moderately sized programs because of the base inefficiency of using only summaries to represent aliasing behavior. As discussed in Section 2, summaries generated in this fashion collectively contain much redundant information, which accumulates up and down the call graph.

Much of this redundancy is due to the function-level representation of aliasing behaviors that are properties of globals and types. To solve this problem our system also infers invariants for globals and types. Consider the following example:

```
struct foo {
    int buf[100];
    int *cur;
};

struct foo *g;
g->cur = &(g->buf);
```

After the assignment the points-to graph contains the pair `((*g).cur, (*g).buf)` indicating that the `cur` field points into the buffer `buf`. Whenever the two labels in a points-to relation share a common prefix ending in a user-defined type (`*g` in this example), we promote the relationship to a global invariant by replacing the prefix by `this` and adding the resulting fact to the invariant map for that type. In this example, we add `(this.cur, this.buf)` to $\rho_{\texttt{foo}}$. Aliasing between globals or globals and types is handled similarly.

As discussed in Section 1, our system can use annotations to tweak the analysis behavior and improve scalability or precision. This offers substantial flexibility in modeling the idiosyncrasies of specific code bases.

4. RESULTS

We have implemented our analysis using framework for summary-based path-sensitive analysis based on Saturn [32]. We run the analysis bottom-up over the call graph, computing a fixed point for strongly connected components, and generate sets of accessed labels, exit aliasing, and inferring new invariant and entry aliasing. This new aliasing may then trigger reanalysis of functions deeper in the call graph. The analysis can be run in parallel, utilizing multiple cores, each core analyzing a single function at a time.

Our aliasing study uses four widely-used applications, ranging in size from 64 KLOC to 575 KLOC, with a total size of 1094 KLOC. We evaluate the results of the analysis based on the number and accuracy of the invariant, entry and exit aliasing generated. In Section 5 we present a detailed taxonomy of aliasing behavior identified in these applications.

Our overall results are shown in Table 1 (all experiments were done on a single machine). In generating summaries for 22283 functions and loops in the applications, we added 945 type invariant points-to edges on a total of 135 types (7.8% of all types) and 569 global invariant edges on a total of 121 globals (5.1% of all globals). For each function in the programs, an average of 0.075 entry aliasing edges and 0.391 exit aliasing edges were generated. Additionally, for each loop in the programs, an average of 0.280 entry aliasing edges and 0.949 exit aliasing edges were generated, significantly higher than for functions.

Counts for types and globals with invariants and functions with entry aliasing are shown in Table 2. Each of these was hand inspected to identify correct aliasing (e.g. aliasing which corresponds to some actual runtime behavior) and *false aliasing* (e.g. added as a result of analysis imprecision), as well as to classify each behavior (see Section 5). Of the 135 composite types with invariant aliasing, 119 have invariants that appear correct, while 27 have invariants that appear false (11 types have both correct and false invariants). Of the 121 global variables with invariant

	Type		Global		Entry		Overall	
Application	T	F	T	F	T	F	T	F
openssh-3.9p1	6	1	11	0	36	1	53	2
openssl-0.9.7e	21	2	18	1	83	10	122	13
httpd-2.0.53	21	7	9	8	27	19	57	34
postgresql-8.0.2	71	17	65	10	220	128	356	155
Overall	119	27	103	19	366	158	588	204

Table 2: True and false aliasing counts for type invariants, global invariants and function entry aliasing. Shown are counts of types/globals/functions where either true or false aliasing was found.

aliasing generated for them, 103 have invariants that appear correct and 19 have invariants that appear false (one global variable has both correct and false invariants). Of the 523 functions with any entry aliasing generated for them, 366 have aliasing that appears correct, while 158 have aliasing that appears false (one function has both correct and false aliasing). Overall, this gives a false aliasing rate of 26.2%, which is sufficiently accurate to make our study possible.

Additional measures relating to the generated exit aliasing give insight not only into how the analysis performs on the analyzed applications, but on how it is likely to perform on larger or more complex ones. We are interested in:

- The amount of aliasing generated. This gives an overall understanding of the overhead required to fully specify the summaries.
- The relationship between the type and global invariants and the amount of per-function aliasing generated. Adding invariants should eliminate redundant entry/exit aliasing and reduce analysis workload.

We performed a second series of runs without generating any type or global invariants, examining the amount of exit aliasing generated. Complexity measures for the runs with and without invariants are shown in Table 3. For the run with invariants, 5900 exit aliasing points-to edges were generated. For the run without invariants, 70735 exit aliasing points-to edges were generated, nearly 12 times as many. The change is most dramatic in PostgreSQL, where not using invariants raises the number of functions with at least 20 exit aliasing edges from 14 to 1799, and raises the maximum number of exit aliasing edges from 34 to 128.

We also consider how the relationship between invariants and generated aliasing on a function is affected by the function's *interface width*, the number of global variables potentially used by the function or its callees. Interface width is a measure of the overall complexity of a function's logic, and generally increases for functions in larger applications and closer to the root of the call graph. If the amount of aliasing generated on a function is independent from the width of that function's interface, then analysis time for a function is affected only by the size of that function's body, and the number of callees it has. As both factors are unrelated to the size of the analyzed application, achieving this independence allows scalability to arbitrarily large applications.

Table 3 includes the R^2 measure of predictive power between interface width and exit aliasing, which indicates the fraction of the variation in exit aliasing on a function explained by that function's interface width. The correlation for the runs with invariants is much weaker than that for the runs without invariants, indicating that with invariants analysis for a function is nearly independent of interface width.

5. AN ALIASING TAXONOMY

The aliasing behavior of an application can be arbitrarily complex, but we find that just a few programming techniques cover almost all cases of aliasing we have observed. We describe a taxonomy for the behavior documented by the invariant and function entry aliasing generated during the runs from Section 4. We refine these behaviors into conceptually related groups, picking out distinguishing characteristics between them.

We are concerned with *data aliasing* behaviors found, which occur when pointers from different parts of the store share references to mutable data. Managing data aliasing is critical to maintaining the organization of an application's data structures, and is the most widespread and diverse aspect of aliasing in these applications. Instances of data aliasing are naturally classified by their relation to unbounded data structures (recursive structures and arrays), which may generally contain any number of elements. Unbounded structures are widespread, and observed aliasing behaviors differ substantially from one another depending on their involvement with these structures:

- *incidental aliasing* occurs when a pointer targets locations outside unbounded structures.
- *cursor aliasing* occurs when a pointer may target locations within an unbounded structure.
- *false aliasing* is data aliasing identified by the analysis, but which corresponds to no runtime behavior.

Additional properties of pointers, such as which may be `NULL`, which may point to (immutable) constant strings or to functions, or which point elsewhere within an unbounded structure, are not considered here, though these are also discovered by our analysis. Table 4 organizes the invariants and function entry aliasing generated. In the following sections we develop finer distinctions within each type.

5.1 Incidental Aliasing

Despite the potential diversity of uses for aliases, almost all incidental aliasing we have observed follows one of five patterns. Incidental aliasing is most often used to provide additional indirection within a heap structure, or for a couple of behaviors on values at the same level of indirection.

- *parent pointers* are references to particular data closer to the root of a structure.
- *child pointers* are additional references to particular data stored deeper in a structure.
- *shared immutable pointers* are multiple references to data at the same level, where all are used only for reading.
- *shared I/O pointers* are two references to data at the same level, where one is used only for reading and the other only for writing.
- *global pointers* are references to a global variable and an alias of the global within the same scope.

Table 5 organizes the incidental aliasing according to these categories.

		With Invariants			Without Invariants		
Application	Functions	$\mathbb{A}_{out}$	$\mathbb{A}_{out}$/Func	R^2	$\mathbb{A}_{out}$	$\mathbb{A}_{out}$/Func	R^2
openssh-3.9p1	1147	145	0.126	0.0006%	1732	1.51	83.2%
openssl-0.9.7e	4109	1662	0.404	0.07%	8918	2.17	35.8%
httpd-2.0.53	1980	1373	0.693	4.33%	2535	1.28	23.8%
postgresql-8.0.2	7842	2720	0.347	0.03%	57550	7.34	78.9%
Overall	15078	5900	0.391		70735	4.69	

Table 3: Amount of function exit aliasing with and without type/global invariants.

	Type				Global				Entry				
Application	Inc	Cur	False	Total	Inc	Cur	False	Total	Inc	Cur	False	Total	Total
openssh-3.9p1	1	5	1	7	8	3	0	11	7	29	1	37	55
openssl-0.9.7e	12	10	2	24	18	1	1	20	67	16	10	93	137
httpd-2.0.53	9	13	7	29	6	3	8	17	18	9	19	46	92
postgresql-8.0.2	35	41	17	93	21	46	10	77	141	81	128	350	520
Overall	57	69	27	153	53	53	19	125	233	135	158	526	804

Table 4: Kinds of type invariants, global invariants, and entry aliasing generated for each application. Counts are numbers of types/globals/functions with the specified behavior.

Parent and child pointers provide additional methods for traversing structures beyond their basic spanning tree. The *depth* of a value is the number of indirections within the tree used to access it. Parent pointers reference data shallower in the tree, and child pointers skip multiple levels deeper. (Recall that parent/child pointers within a homogeneous recursive data structure are collapsed to a single unbounded location (Section 3.4); thus, in the parent/child aliasing patterns we discover the source and target of the pointer are of different types.)

Example 3.

`httpd-2.0.53/include/httpd.h`: Structure `conn_rec` holds per-connection information. Its field `remote_host` may reference the `hostname` of the `remote_addr` field, if a DNS lookup has been performed. [Figure 2]

`postgresql-8.0.2/src/include/lib/dllist.h`: Structure `Dlelem` is a doubly linked list element with a pointer field `dle_list` to its parent.

`postgresql-8.0.2/src/bin/pg_dump/pg_backup_db.c`: Function `_connectDB` takes strings `reqdb` and `reqname`, which may be internal to the archive handle `AH` passed into it as well. □

If two potentially aliased values are at the same depth in the heap, there is generally no clear precedence between them. We find that such behaviors are described almost entirely by two patterns, shared immutable and I/O aliasing.

Shared immutable aliasing occurs when neither of the two values are used for writing. Since reads on aliased values cannot affect each other, such aliasing is benign.

Example 4.

`openssl-0.9.7e/crypto/x509/x509_vfy.c`: Function `check_issued` checks whether certificate `x` was issued by `issuer`. `x` and `issuer` may be identical, but neither is written to.

`openssh-3.9p1/packet.c`: Global structures `receive_context` and `send_context` may share the same read-only field `cipher`, indicating how incoming and outgoing data is encrypted. □

Shared I/O aliasing occurs when one of two aliased values is used for reading and the other for writing. This programming model is very flexible, allowing space reuse only where

include/httpd.h:

```
/** Structure to store things which are per connection */
struct conn_rec {
    /** remote address */
    apr_sockaddr_t *remote_addr;

    /** Client's DNS name, if known. ...
     * Only access this though get_remote_host() */
    char *remote_host;
}
```

server/core.c:

```
const char * ap_get_remote_host(conn_rec *conn)
{   // aliasing is introduced by this function call
    if (apr_getnameinfo(&conn->remote_host, conn->remote_addr)
        == APR_SUCCESS) {
        ...
    }
    if (conn->remote_host != NULL) {
        return conn->remote_host;
    }
}
```

srclib/apr/network_io/unix/sockaddr.c:

```
apr_status_t apr_getnameinfo(char **hostname,
                             apr_sockaddr_t *sockaddr)
{
    char tmphostname[256];
    ...
    *hostname = sockaddr->hostname = apr_pstrdup(tmphostname);
    return APR_SUCCESS;
}
```

Figure 2: Child pointer example from httpd-2.0.53.

crypto/aes/aes_core.c:

```
/* Encrypt a block; in and out can overlap */

void AES_encrypt(const u_char *in, u_char *out,
                 const AES_KEY *key) {
    s0 = GETU32(in     );
    s1 = GETU32(in +  4);
    s2 = GETU32(in +  8);
    s3 = GETU32(in + 12);
    ...
    PUTU32(out     , t0);
    PUTU32(out +  4, t1);
    PUTU32(out +  8, t2);
    PUTU32(out + 12, t3);
}
```

crypto/aes/aes_cbc.c:

```
void AES_cbc_encrypt(const u_char *in, u_char *out,
                     u_long len, const AES_KEY *key,
                     u_char *ivec, const int enc) {
    u_char tmp[AES_BLOCK_SIZE];
    ...
    if (len) {
        for(n=0; n < len; ++n)
            tmp[n] = in[n] ^ ivec[n];
        for(n=len; n < AES_BLOCK_SIZE; ++n)
            tmp[n] = ivec[n];
        AES_encrypt(tmp, tmp, key); // aliasing introduced
        memcpy(out, tmp, AES_BLOCK_SIZE);
    }
}
```

Figure 3: Shared I/O example from openssl-0.9.7e.

desired. However, care must be taken when writing code that uses I/O aliasing to ensure that the input data is never corrupted while writing.

Example 5.

`openssl-0.9.7e/crypto/aes/aes_core.c`: Function `AES_encrypt` encrypts the block of data at `in`, storing the result at `out`. `in` and `out` may overlap. [Figure 3]

`postgresql-8.0.2/src/backend/utils/adt/numeric.c`: Function `add_var` (and several others) takes arguments `var1`, `var2`, and `result`, storing the sum of `var1` and `var2` in `result`. `result` may be the same as either `var1` or `var2`. □

Global aliasing on a function occurs when a pointer to global data is passed to that function, but the function may also directly access the global itself. Often this aliasing is innocuous; neither pointer is written, or the global is only accessed when the local pointer is `NULL`. However, some uses require care similar to what is needed when using shared I/O.

Example 6.

`openssl-0.9.7e/apps/apps.c`: Function `load_cert` writes logs to both its parameter `err` and global variable `bio_err`, which are typically aliased.

`postgresql-8.0.2/src/include/executor/execdesc.h`: Field `dest` of type `QueryDesc` references a table of function pointers to call with the results of an SQL query. It typically references one of three tables in `src/backend/tcop/dest.c`, either `donothingDR`, `debugtupDR`, or `spi_printtupDR`. □

Of the remaining miscellaneous instances of incidental aliasing, most seem to describe very specialized aliasing behaviors. We have not investigated the sources of all these instances. However, some appear either unexpected or, at the least, unpredictable and potentially dangerous. One instance in PostgreSQL was confirmed as a bug by the application's developers:

src/bin/pg_dump/pg_backup_archiver.c:

```
void SortTocFromFile(ArchiveHandle *AH, RestoreOptions *ropt) {
    /* Set prev entry as head of list */
    TocEntry *te, *tePrev = AH->toc;

    fh = fopen(ropt->tocFile, PG_BINARY_R);
    while (fgets(buf, 1024, fh) != NULL) {
        /* Get an ID */
        id = strtol(buf, &endptr, 10);

        /* Find TOC entry */
        te = getTocEntryByDumpId(AH, id);
        if (!te)
            die_horribly("could not find entry for ID \%d\n",id);

        _moveAfter(AH, tePrev, te); // tePrev & te may be aliased
        tePrev = te;
    }
    fclose(fh);
}

TocEntry* getTocEntryByDumpId(ArchiveHandle *AH, DumpId id) {
    TocEntry *te = AH->toc->next;
    while (te != AH->toc) {
        if (te->dumpId == id)
            return te;
        te = te->next;
    }
    return NULL;
}

void _moveAfter(ArchiveHandle *AH, TocEntry *pos, TocEntry *te) {
    te->prev->next = te->next;
    te->next->prev = te->prev;
    te->prev = pos;
    te->next = pos->next;
    pos->next->prev = te;
    pos->next = te;
}
```

Figure 4: Aliasing bug from postgresql-8.0.2.

Example 7.

`postgresql-8.0.2/src/bin/pg_dump/pg_backup_archiver.c`: Function `SortTocFromFile` sorts a ring of `TocEntry` values according to a list of IDs read from disk, calling `_moveAfter` successively on pairs of ring values. If the same ID is read twice in a row, `_moveAfter` is called with aliased values, corrupting the ring. [Figure 4] □

5.2 Cursor Aliasing

Cursors are used to organize and manage the contents of unbounded structures. Cursors require *indexes*, ways to query or traverse their contents. We classify instances of cursor aliasing by their relation to the indexes in the structure they reference.

- *index cursors* support the use of an additional index for a structure.
- *tail cursors* hold the end point of an existing index.
- *query cursors* read data internal to an existing index.
- *update cursors* write data internal to an existing index.

Table 6 organizes the cursor aliasing according to these categories.

ssh-keyscan.c:

```
/* Keep a connection structure for each file descriptor.  The state
 * associated with file descriptor n is held in fdcon[n].
 */
typedef struct Connection {
    ...
      /* Hostname of connection for errors */
    char *c_name;
      /* Hostname of connection for output */
    char *c_output_name;
      /* Quick lookup: c->c_fd == c - fdcon */
    int c_fd;
      /* List of connections in timeout order. */
    TAILQ_ENTRY(Connection) c_link;
} con;

/* Timeout Queue */
TAILQ_HEAD(conlist, Connection) tq;
con *fdcon;

int conalloc(char *iname, char *oname, int keytype) {
    char *name = xstrdup(iname);
    s = tcpconnect(name);

    fdcon[s].c_fd = s;
    fdcon[s].c_name = name;
    fdcon[s].c_output_name = xstrdup(oname);
    TAILQ_INSERT_TAIL(&tq, &fdcon[s], c_link); // aliasing introduced
    return (s);
}

void confree(int s) {
    xfree(fdcon[s].c_name);
    xfree(fdcon[s].c_output_name);
    TAILQ_REMOVE(&tq, &fdcon[s], c_link);
}

void contouch(int s) {
    TAILQ_REMOVE(&tq, &fdcon[s], c_link);
    TAILQ_INSERT_TAIL(&tq, &fdcon[s], c_link);
}
```

Figure 5: Example index cursor from openssh-3.9p1.

Every unbounded structure must have a *primary* index; there must be some way to retrieve or update the stored data. Index cursors enable the use of additional indexes. There are several ways index cursors can be used:

Example 8.
`openssh-3.9p1/ssh-keyscan.c`: `fdcon` is an array mapping file descriptor IDs to connections. `tq` is a timeout queue whose members are entries of `fdcon`. [Figure 5]
`postgresql-8.0.2/src/backend/storage/freespace/freespace.c`: `FSMHeader` is a structure with two lists `firstRel` and `usageList`, which index the same set of relations by storage order and last access order, respectively.
`postgresql-8.0.2/src/bin/pg_dump/common.c`: `dumpIdMap` is an array mapping dump IDs to a corresponding dumpable object. The array `catalogIdMap` is generated from `dumpIdMap`; the entries are cursors to values in `dumpIdMap`. □

If an index has a linear structure (there is no branching in its traversal) a tail cursor can mark its end point.

Example 9.
`openssl-0.9.7e/crypto/engine/eng_list.c`: `engine_list_tail` is a cursor to the last value in the doubly linked list referenced by `engine_list_head`.
`openssl-0.9.7e/ssl/ssl_ciph.c`: `ssl_cipher_apply_rule` takes a cipher list; the head is `*head_p`; the tail is `*tail_p`. □

src/backend/nodes/list.c:

```
/* Delete 'cell' from 'list'; 'prev' is the previous element to 'cell'
 * in 'list', if any (i.e. prev == NULL iff list->head == cell)
 */
List* list_delete_cell(List *list, ListCell *cell, ListCell *prev) {
    if (prev)
        prev->next = cell->next;
    else
        list->head = cell->next;

    if (list->tail == cell)
        list->tail = prev;

    pfree(cell);
    return list;
}

/* Delete the first element of the list. */
List* list_delete_first(List *list) {
    return list_delete_cell(list, list->head, NULL);
}
```

Figure 6: Update cursor from postgresql-8.0.2.

Index and tail cursors together form 16.4% of all cursor aliasing behavior found. The remainder may reference any data accessible via a structure's indexes and supply the principal means by which the structure is accessed by the application. These other cursors are grouped into query cursors and update cursors, which are used, respectively, to support read and read/write operations on the structure. Understanding the mechanisms by which programmers manage separate, potentially interfering cursors to the same unbounded structure is an interesting area for future study.

Example 10.
`postgresql-8.0.2/src/backend/nodes/list.c`: `list_delete_cell` takes a singly linked `list` and two list elements `cell` and `prev`, which must be adjacent and inside `list`. [Figure 6]
`httpd-2.0.53/srclib/apr/tables/apr_hash.c`: The `apr_hash_t` structure contains an array of hash elements `array` and an `iterator`, whose `this` field points into the outer `array` and may be used to update it.
`postgresql-8.0.2/src/backend/storage/buffer/bufmgr.c`: Parameter `buf` of `FlushBuffer` must point to an element of the global `BufferDescriptors` array. □

5.3 False Aliasing

Table 7 organizes the instances of false aliasing according to the behavior leading to the misidentification.

The great majority of false aliasing is due to the use of disjoint parts of unbounded data structures. If multiple pointers to values within an unbounded structure are created and then accessed together, the pointers are identified as aliased regardless of whether they must refer to disjoint parts of the unbounded structure. Determining that values are disjoint can require shape or path information, but most commonly requires separation of individual array indices. One case in particular, a parameter passing `VARARGS` structure in PostgreSQL which contains an array of function arguments, is responsible for over 1/3 of all false aliasing discovered.

Other sources of false aliasing more specifically involve shape and path information. Several instances are due to moving values from one unbounded structure to another, often via free lists maintained by random-access structures.

App/Kind	Par	Chi	Imm	I/O	Glob	Misc	Total
openssh			4	4		7	15
openssl	3	10	26	39	16	20	114
httpd	3	2	9		3	13	30
postgresql	15	35	25	13	82	35	205
Type	21	13	14	1	13	9	71
Global		2	2			55	59
Function		32	48	55	88	11	234
Overall	21	47	64	56	101	75	364

Table 5: Incidental aliasing kinds.

App/Kind	Index	Tail	Query	Update	Total
openssh	1	8	2	26	37
openssl	3	6	12	10	31
httpd	3	6	9	8	26
postgresql	3	14	120	38	175
Type	6	17	41	16	80
Global	3	3	38	9	53
Function	1	14	64	57	136
Overall	10	34	143	82	269

Table 6: Cursor aliasing kinds.

App/Kind	Disjoint	Varargs	Moving	Paths	Total
openssh	1			1	2
openssl	10			3	13
httpd	15		6	13	34
postgresql	56	70	3	26	155
Type	10		3	14	27
Global	1		1	17	19
Function	71	70	5	12	158
Overall	82	70	9	43	204

Table 7: False aliasing kinds.

The remaining false aliasing originates from path-sensitive behavior across functions. Typically, two side effects of a function are coordinated (e.g., appending one list to another and NULL-ing out the original); by merging them together, spurious behavior is modeled and false aliasing results.

6. RELATED WORK

We briefly survey a wide range of related work on the evaluation of alias analysis and alias control. Previous studies of alias analysis on large programs either statically compare the precision of several pointer analyses [14, 27, 28, 12] or use dynamic analysis to evaluate the precision of pointer analyses [25, 21]. Among the dynamic analysis studies, Mock et al. [25] find that several context-insensitive analyses heavily over-approximate the points-to sets observed during a typical run, while Liang et al. [21] find that, for Java programs, context-insensitive analyses handle most allocation sites and programs very well, but are very imprecise for others. Another study directly compares context-sensitive and context-insensitive points-to analysis for Java and confirms that context-sensitivity is much more precise in the sense that each individual context has very few points-to relationships compared to the flow-insensitive analysis' summary over all contexts [29]. Such empirical comparisons of pointer analyses are critical to evaluate the effectiveness of design decisions as such as context-sensitivity vs. context-insensitivity. However, the metrics used, generally points-to set sizes and related properties such as side effect sizes or value liveness, are not useful for understanding how aliasing is used. Earlier studies of aliasing structure focus on heap shape properties and look at smaller programs (e.g., [13]).

Most recent scalable alias analyses use a flow-insensitive intraprocedural core on which a context- or object-sensitive analysis is built [3, 19, 29]. Our flow- and path-sensitive intraprocedural analysis is different, harking back to early work on aliasing; for example, Landi and Ryder introduced labels for naming access paths fifteen years ago [18]. One other recent work also uses a flow-, path-, and context-sensitive analysis, showing good results for programs up to 13,000 LOC [22]; unfortunately, this system is not available and we were unable to perform an empirical comparison.

Our technique for object naming is more general than previous work, which typically selectively inlines a few allocation functions to generate more allocation points (i.e., object names). Our object naming scheme achieves benefits roughly similar to unlimited object-sensitivity in object-oriented languages [23], because object-sensitivity focuses on providing distinctions associated with data based on allocation site rather than context-sensitivity, which provides distinctions based on calling context.

Some common characteristics of program aliasing behavior have been exploited by previous pointer analyses. Das [9] uses the fact that pointers are commonly used for passing addresses of stack objects in C. Aiken et al. [1] find that many locks are *restricted*: while multiple aliases may exist, only one is used within a scope (e.g., a function body), allowing precise flow-sensitive analysis. Our study shows that almost all values are used in this same restricted fashion.

Several previous points-to analyses are summary-based. Liang and Harrold [20] separate local and global pointers to achieve a summary-based alias analysis that scales well, but is also unification-based and flow-insensitive. Some previous context-sensitive approaches use a similar summary to ours (input and output points-to graphs) with similar issues (e.g., the mapping between caller/callee names) [18, 10, 6]. For more expressive approaches the summaries appear to grow excessively large for large programs [31, 30]; we avoid this problem by exploiting heap invariants to compactly represent global information and simplify summaries.

Alias control techniques specify the presence or absence of aliasing in programs. Uniqueness-based techniques describe how and where pointers are unaliased [24, 4]. We believe the linear nature of unique pointers imposes unnecessary restrictions: it is common to copy data and have both values be live (through function calls, local variables, etc.), though in separate scopes. Fähndrich and Deline [11] allow aliasing on linear values by restricting how the aliases are used.

Ownership-based techniques constrain the possible aliases of a pointer. Early ownership techniques include Hogg's Islands [15], which restrict external aliases to the entire contents of particular objects. Later work has made ownership systems more flexible by having multiple ownership domains for the contents of an object [8] and adding access permissions between separate domains [17].

Aldrich et al. introduce AliasJava [2], which adds both uniqueness and ownership annotations to document aliasing behavior in Java. While uniqueness allows AliasJava to cleanly describe unaliased pointers, we believe the mechanisms offered by ownership are insufficient to accurately describe actual aliasing. Aliased pointers are either owned, and may be aliased with any other pointer with the same owner, or are shared and may be aliased with any pointer.

In contrast, the annotations used and constraints generated by our analysis directly specify the targets of aliased pointers, correlating closely to runtime aliasing behavior without imposing significant documentation overhead.

7. CONCLUSION

We have developed a new context-, flow-, and partially path-sensitive alias analysis, which we have used to identify all sources of aliasing in over one million lines of C code. A distinctive feature of our approach is the use of type and global invariants to avoid redundant computation of widespread aliasing. We have found aliasing has a great deal of structure in real programs and that just nine programming idioms account for nearly all aliasing in our study.

8. REFERENCES

[1] A. Aiken, J. Foster, J. Kodumal, and T. Terauchi. Checking and inferring local non-aliasing. In *Proc. of the Conference on Programming Language Design and Implementation*, pages 129–140, 2003.

[2] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Proc. of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 311–330, 2002.

[3] M. Berndl, O. Lhotak, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *Proc. of the Conference on Programming Language Design and Implementation*, pages 103–114, 2003.

[4] J. Boyland. Alias burying: Unique variables without destructive reads. *Softw. Pract. Exper.*, 31(6):533–553, 2001.

[5] D. Chase, M. Wegman, and K. Zadeck. Analysis of pointers and structures. In *Proc. of the Conference on Programming Language Design and Implementation*, pages 296–310, 1990.

[6] R. Chatterjee, B. Ryder, and W. Landi. Relevant context inference. In *Proc. of the Symposium on Principles of Programming Languages*, pages 133–146, 1999.

[7] J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proc. of the Symposium on Principles of Programming Languages*, pages 232–245, 1993.

[8] D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proc. of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 48–64, October 1998.

[9] M. Das. Unification-based pointer analysis with directional assignments. In *Proc. of the Conference on Programming Language Design and Implementation*, pages 35–46, 2000.

[10] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proc. of the Conference on Programming Language Design and Implementation*, pages 242–256, 1994.

[11] M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proc. of the Conference on Programming Language Design and Implementation*, pages 13–24, 2002.

[12] J. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Proc. of the International Symposium on Static Analysis*, pages 175–198, 2000.

[13] R. Ghiya and L. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Proc. of the Symposium on Principles of Programming Languages*, pages 1–15, 1996.

[14] M. Hind and A. Pioli. Which pointer analysis should I use? In *Proc. of the International Symposium on Software Testing and Analysis*, pages 113–123, 2000.

[15] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *Proc. of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 271–285, 1991.

[16] N. Jones and S. Muchnick. Flow analysis and optimization of lisp-like structures. In *Program Flow Analysis: Theory and Applications*, pages 102–131. Prentice Hall, 1979.

[17] N. Krishnaswami and J. Aldrich. Permission-based ownership: Encapsulating state in higher-order typed languages. In *Proc. of the Conference on Programming Language Design and Implementation*, June 2005.

[18] W. Landi and B. Ryder. A safe approximate algorithm for interprocedural aliasing. In *Proc. of the Conference on Programming Language Design and Implementation*, pages 235–248, 1992.

[19] O. Lhotak and L. Hendren. Jedd: a BDD-based relational extension of Java. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 158–169, 2004.

[20] D. Liang and M. J. Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. In *Proc. of the International Symposium on Static Analysis*, pages 279–298, 2001.

[21] D. Liang, M. Pennings, and M. Harrold. Evaluating the precision of static reference analysis using profiling. In *Proc. of the International Symposium on Software Testing and Analysis*, pages 22–32, 2002.

[22] V. B. Livshits and M. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. In *Proc. of the Symposium on the Foundations of Software Engineering*, pages 317–326, 2003.

[23] A. Milanova, A. Rountev, and B. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proc. of the International Symposium on Software Testing and Analysis*, pages 1–11, 2002.

[24] N. Minsky. Towards alias-free pointers. In *Proc. of the European Conference on Object-Oriented Programming*, pages 189–209, 1996.

[25] M. Mock, M. Das, C. Chambers, and S. Eggers. Dynamic points-to sets: A comparison with static analyses and potential applications in program understanding and optimization. In *Proc. of the Workshop on Program Analysis for Software Tools and Engineering*, pages 66–72, 2001.

[26] M. Naik and A. Aiken. Effective static race detection for Java. In *Proc. of the Conference on Programming Language Design and Implementation*, page to appear, 2006.

[27] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proc. of the Conference on Programming Language Design and Implementation*, pages 13–22, 1995.

[28] P. Stocks, B. Ryder, W. Landi, and S. Zhang. Comparing flow and context sensitivity on the modification-side-effects problem. In *Proc. of the International Symposium on Software Testing and Analysis*, pages 21–31, 1998.

[29] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proc. of the Conference on Programming Language Design and Implementation*, pages 134–144, 2004.

[30] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. *Proc. of the Conference on Object Oriented Programming Systems, Languages, and Applications*, pages 187–206, 1999.

[31] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proc. of the Conference on Programming Language Design and Implementation*, pages 1–12, 1995.

[32] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *Proc. of the Symposium on Principles of Programming Languages*, pages 351–363, January 2005.