Analyzing Memory Ownership Patterns in C Libraries *

Tristan Ravitch
Department of Computer Sciences
University of Wisconsin–Madison
travitch@cs.wisc.edu

Ben Liblit
Department of Computer Sciences
University of Wisconsin–Madison
liblit@cs.wisc.edu

Abstract

Programs written in multiple languages are known as *polyglot programs*. In part due to the proliferation of new and productive high-level programming languages, these programs are becoming more common in environments that must interoperate with existing systems. Polyglot programs must manage resource lifetimes across language boundaries. Resource lifetime management bugs can lead to leaks and crashes, which are more difficult to debug in polyglot programs than monoglot programs.

We present analyses to automatically infer the ownership semantics of C libraries. The results of these analyses can be used to generate bindings to C libraries that intelligently manage resources, to check the correctness of polyglot programs, and to document the interfaces of C libraries. While these analyses are unsound and incomplete, we demonstrate that they significantly reduce the manual annotation burden for a suite of fifteen open source libraries.

Categories and Subject Descriptors D.2.11 [*Software Engineering*]: Software Architectures—Languages; D.2.12 [*Software Engineering*]: Interoperability; D.2.13 [*Software Engineering*]: Reusable Software—Reusable libraries; D.3.2 [*Programming Languages*]: Language Classifications—C, Python; D.3.3 [*Programming Languages*]: Language Constructs and Features—Dynamic storage management, Procedures, functions, and subroutines; D.3.4 [*Programming Languages*]: Processors—Code generation, Memory management; E.1 [*Data Structures*]: Arrays, Records; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

Keywords resource lifetime management; memory allocation; allocators; finalizers; ownership transfer; escape analysis; sharing; reference counting; foreign function interfaces (FFIs); bindings; libraries; dataflow analysis; interprocedural static program analysis; polyglot programming; interoperability

```
char *strdup(const char *s);
char *asctime(const struct tm *tm);
```

Figure 1. C function signatures

1. Introduction

High-level programming languages have been gaining acceptance in many application domains where unsafe low-level languages like C and C++ were once the only option. For example, Python and JavaScript have a significant presence in the desktop application space. Additionally, Python has gained acceptance in the scientific computing community [14]. Unfortunately, these high-level languages do not exist in a vacuum. They depend on code written in unsafe lower-level languages for, among other reasons, performance and interoperability. Some performance-sensitive pieces of code simply cannot be rewritten in the desired high-level language. In other cases, it is possible but not economically feasible to rewrite working and tested code, so the original implementation must be used from the high-level language.

Production-quality high-level languages support calling functions from libraries written in other languages through foreign function interfaces (FFIs). High-level language programs using FFIs execute code from more than one language, making them *polyglot programs*. A critical challenge in writing correct polyglot programs lies in managing the flow of resources across language boundaries. Programs lacking a precise cross-language ownership semantics are vulnerable to resource leaks, threatening reliability. Unclear ownership semantics can also lead to crashes induced by use-after-free or double-free errors. Both of these types of errors are more difficult to debug in polyglot programs, as common debugging tools target programs written in a single language.

Most low-level languages like C do not provide any means for describing, much less checking, object ownership semantics. Instead, this critical information must be conveyed through documentation or recovered through static analysis. Library interfaces defined in C typically refer to dynamically-allocated resources by their address (a pointer). Unfortunately one cannot simply call free on all pointers obtained from a low-level language to release the associated resources, or *finalize* them. This approach fails because, while C functions expose most resources through pointers, they use pointers for many other purposes as well.

Consider the two C function declarations in figure 1. Each returns a **char***. The value returned by strdup must be freed to prevent memory leaks, but freeing the value returned by asctime will cause a crash. The caller owns the result of strdup but not the result of asctime. Furthermore, some dynamically-allocated resources may require specialized finalizer functions instead of generic free. Consider fopen and fclose: calling free on the result of fopen is a partial resource leak. Thus, functions returning managed resources must be identified along with their associated resource finalizers.

* Supported in part by DoE contract DE-SC0002153, LLNL contract B580360, NSF grants CCF-0953478 and CCF-1217582, and a grant from the Wisconsin Alumni Research Foundation. Opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of NSF or other institutions.

ISMM'13 June 20–21, 2013, Seattle, Washington, USA
Copyright © 2013 ACM 978-1-4503-2100-6/13/06. . . $15.00
Reprinted from ISMM'13, Proceedings of the 12th International Symposium on Memory Management, June 20–21, 2013, Seattle, Washington, USA, pp. 1–11.

```
typedef struct pvl_elem_t {
  void *data;
  struct pvl_elem_t *next;
} pvl_elem;

typedef struct pvl_list_t {
  pvl_elem *head;
} pvl_list;

typedef struct icalcomponent {
  pvl_list *components;
  struct icalcomponent* parent;
} icalcomponent;

void pvl_push(pvl_list *lst, void *d) {
  pvl_elem *e = calloc(1, sizeof(pvl_elem));
  e→next = lst→head;
  lst→head = e;
  e→data = d;
}

void* pvl_pop(pvl_list *lst) {
  if(lst→head == NULL) return NULL;
  list_elem *e = lst→head;
  void *ret = e→data;
  lst→head = e→next;
  free(e);
  return ret;
}
```

Figure 2. Definitions for figure 3

Resource management semantics are unclear in C even without the additional complexities of polyglot programming.

We describe our ownership model for C resources and present algorithms to infer the ownership semantics of C libraries. These semantics are presented to users and tools through *inferred annotations* on library functions. While these analyses are unsound and incomplete, they are nonetheless useful. As discussed in section 8, our algorithms significantly reduce the *manual* annotation burden required to create library bindings. With review by a programmer familiar with the library being analyzed, these inferred annotations are sufficient to generate idiomatic FFI library bindings for high-level languages. The resulting bindings will be idiomatic in that they *automatically* manage the flow of resources between languages and clean them up when they become garbage. They also serve as an aid in program understanding and can augment documentation. Our primary contributions over prior work are: (1) two methods for identifying the ownership transfers of objects in C libraries (sections 4 and 5), (2) a method for inferring contracts that must be obeyed for function pointers (section 6), and (3) an algorithm for identifying reference-counted resources (section 7).

2. Resource Ownership Model

We adopt the ownership model of Heine and Lam [6] whereby each object is pointed to by exactly one owning reference. The object must eventually either be *finalized* through the owning reference, or ownership must be transferred to another owning reference. When a pointer is finalized, the resources held by the object it points to are safely released. Non-owning references to any object can be created at any time and are valid until the object is finalized. In the Heine and Lam model, pointer-typed members of C++ objects are either always owning references or are never owning references (at public interface boundaries). We extend this model to pointer-typed fields of some C structures for all functions.

```
1  struct icalcomponent* icalcomponent_new() {
2    icalcomponent* comp = malloc(sizeof(icalcomponent));
3
4    if (!comp) return NULL;
5
6    comp→components = newlist();
7    comp→parent = NULL;
8
9    return comp;
10 }
11
12 void icalcomponent_free(icalcomponent* c) {
13   icalcomponent* comp;
14
15   if (!c) return;
16
17   while ((comp=pvl_pop(c→components)) != NULL) {
18     icalcomponent_remove_component(c,comp);
19     icalcomponent_free(comp);
20   }
21
22   pvl_free(c→components);
23   free(c);
24 }
25
26 void icalcomponent_remove_component(
27   icalcomponent *component, icalcomponent *child);
28
29 void icalcomponent_add_component(icalcomponent *c,
30     icalcomponent *child) {
31   pvl_push(c→components, child);
32 }
33
34 void icalcomponent_set_parent(icalcomponnent *c,
35     icalcomponent* parent) {
36   c→parent = parent;
37 }
```

Figure 3. Example from ical

2.1 Memory Ownership in C

A memory allocator is an abstraction over the most prevalent resource in most programs: dynamically allocated memory. The standard C library's allocator and finalizer functions are malloc and free, respectively. When the memory allocator owns a piece of memory (i.e., the memory is unallocated), it is an error for any other part of the program to use it. When the allocator function is called, it completely transfers ownership of the memory to the caller via the returned owning reference.

Complex resources may own other resources through owned pointer fields. Finalizers for these complex resources must finalize their owned resources to obey the ownership model and to avoid leaks, as in figures 2 and 3. The icalcomponent type is a resource allocated with icalcomponent_new. It owns a component list, along with each of the components in the list of children. The finalizer for this type, icalcomponent_free, finalizes the list of children as well as the child components before finalizing the component itself with a call to free on line 23.

Ownership extends beyond just allocators and finalizers. For example, the function icalcomponent_remove_component removes a child component from a component without finalizing it.

```
with pinned(r):
    # allow r to escape into a global
    stash_in_global(r)

    # drop explicit reference to r, but is still pinned
    del r

    # r is pinned, so safe to access via stashed global
    use_stashed_global()
```

Figure 4. Pinning Python objects with a context manager

After a call to this function, the component no longer owns the child and ownership is implicitly transferred to the caller. Note that simply reading a child component from a component does not transfer ownership because the component will still finalize all of its children when it is itself finalized. Similarly, components do not own the component referenced by their parent field because that field is not finalized in the component finalizer. Our analysis does not automatically recognize this type of ownership transfer. It is relatively rare in real code, and would require expensive shape analysis [18].

2.2 Ownership in High-Level Languages

When a C allocator is called from a high-level language, the high-level language run-time system assumes exclusive ownership of the allocated resource by wrapping it in a special object. All references to the C resource in the high-level language are mediated through this wrapper object, which is managed by the high-level language memory manager (i.e., garbage collector). Since the wrapper object is a normal high-level language object, the memory manager knows when it is unreferenced and safe to finalize. The wrapper object uses memory manager hooks to *automatically* invoke the appropriate finalizer for the C resource when doing so is safe. In contrast, there is no such system in C unless it is implemented manually, such as through reference counting.

Of course, to be of any use these resources must be passed back to low-level code, in which operations on them are written. For a low-level language resource r owned exclusively by a high-level language run-time, assume that it is passed to a low-level language function f: $f(\ldots, r, \ldots)$. For each such call, one of the following must hold:

1. f assumes ownership of r. The high-level language must relinquish ownership of r by disabling any garbage collector hooks that would have run a finalizer on r.
2. f creates only transient references to r, all of which are destroyed when f returns. The high-level language still owns r and need take no further actions.
3. f creates a non-transient non-owning reference n to r. The high-level language run-time system still owns r and does not need to take any further actions. However, the programmer passing r to f must ensure that the lifetime of r exceeds that of n.
4. r does not obey our ownership model, but is instead reference-counted. In this case, as long as the reference manipulation functions are known, the object can safely be passed between languages.
5. r does not obey our ownership model and its resources cannot be automatically managed by the high-level language.

Cases one, two, and four can be fully automated and are ideal for robust language interoperability. The third case requires the high-level language caller of f to understand the semantics of the called function and its effect on the lifetime of r. Note that this semantic knowledge is required of any caller of f, even in C. While the lifetime management of r in the third case cannot be fully automated, a high-level language library binding could provide programmers with tools to make such lifetime management simpler. For example, a Python library binding could provide a resource manager to pin objects to keep them alive within a lexical scope, as in figure 4. In this example, assume that stash_in_global(r) lets r escape into a global location managed by the library. If use_stashed_global accesses r through that global location, then r must still be live when use_stashed_global is called. The pinned resource manager retains a reference to its argument for the lexical scope of the **with** statement.

3. Allocators and Finalizers

Prior work in automating language interoperability by Ravitch et al. [16] presented algorithms for identifying the functions comprising memory allocators. We characterize the memory allocators identified by this work as *derived allocators* because they are built on top of lower-level allocators, with the standard allocator malloc as the ultimate base.

This prior work identifies a function as an allocator if it always returns the result of a base allocator (such as malloc) and gives up ownership of the allocation or returns NULL (to report failure). This is a *must* analysis: it identifies functions that must return a new resource. A must analysis is appropriate here because it can only miss allocator functions. At worst this can cause a leak (or require explicit resource-release calls). By contrast, over-approximating the set of allocators could lead to crashes. In figure 3, icalcomponent_new is an allocator because it returns NULL on line 4 and the result of a call to malloc on line 9.

The corresponding finalizer for an allocator f is a function that takes an argument that is the same type as the return value of f and, on every path, that argument is NULL or is finalized. Ravitch et al. under-approximate the set of finalizers with dataflow analysis. The example in figure 3 shows a finalizer: icalcomponent_free. On one path (at line 15), the argument c is NULL and on the other path it is finalized (at line 23).

A key feature of derived allocators is that they obtain a resource from a lower-level allocator and completely transfer ownership of it to their caller. While most allocation functions in libraries are derived allocators, some libraries define custom memory allocators based on pools, arenas, or some other abstraction. We require manual annotations to identify these custom low-level allocators. This burden is low because few libraries define their own allocators. Additionally, a manual annotation for the allocator and its associated finalizer of a custom allocator allows the analysis to identify many derived allocators.

In principle, shape analysis could identify even these custom allocators. We do not attempt this due to the scarcity of custom allocators and the relative expense of the required analysis.

4. Recognizing Ownership Transfer

As discussed in section 2.2, knowledge of the ownership model of a resource allows a high-level language library binding to automatically manage its lifetime. A key aspect of the ownership model is recognizing which library functions transfer ownership of resources; this allows the high-level language run-time system to assume responsibility for managing library resources safely. Prior work by Ravitch et al. [16] used an escape analysis to conservatively identify library functions that take ownership away from the caller. Intuitively, when a library function causes a pointer provided by the high-level language caller to escape, the lifetime of the pointed-to object becomes unknown and the high-level language run-time system can never safely finalize it.

Certainly if a parameter does not escape, then its ownership is not transferred. Escaping allows transfer, but does not necessarily lead to transfer in all cases. For example, common container-like data structures in C (e.g., lists, trees, or hash tables) typically store pointers to data objects. Clearly, storing an object in one of these containers allows it to escape. However, most of these containers do not take ownership of their elements: the caller must still manage their memory. Ma and Foster [10] note that ownership transfer in library interfaces is relatively rare; we note that escaping parameters are anything but rare. We therefore suggest that escape analysis is distinct from ownership transfer analysis. In this section, we present a transfer analysis to more accurately identify ownership transfers in C library functions. We revisit escape analysis in section 5.

The essence of our ownership transfer analysis follows from our resource ownership model in section 2. We consider a structure field to be owned if that field will be finalized within the context of a finalizer function. Thus, ownership of an object is transferred from a caller when another object assumes responsibility for finalizing it (i.e., it is stored into a field that will be finalized). Fields finalized in other contexts are sometimes used to store temporary allocations, but these fields are not guaranteed to be finalized with the rest of the object and are managed separately. The analysis proceeds in two phases. First it identifies all of the *owned fields* in a library. Then it identifies all of the function parameters that may be stored into an owned field. We describe fields in terms of *symbolic access paths*. Note that we assume that any given field is either always owned or never owned. We do not attempt to reason about fields that are only owned sometimes. Furthermore, we assume that, if one element held in a container-like field (i.e., an array or linked data structure) is owned, all elements in that field are owned.

4.1 Symbolic Access Paths

This analysis is based on symbolic access paths [2, 7]; we follow the formulation of Matosevic and Abdelrahman [11]. An *access path* describes a memory location accessible from a base value by a (possibly empty) sequence of path components. While we will only discuss field accesses in this paper, pointer dereferences, array accesses, and union accesses are also valid path components. We treat all array elements in a single array as identical; a more precise analysis could differentiate between them. This treatment of field accesses is unsound when pointers to **struct** types are cast to unrelated types [15], which could cause the analysis to identify invalid ownership transfers.

Let $ap(v)$ represent the access path for a source expression v. For example, the assignment on line 18 of figure 3 assigns a value to lst→head. The corresponding access path $ap(\text{lst}\rightarrow\text{head})$ is the pair $(\text{lst}, \langle\text{head}\rangle)$. In this pair, lst is the base value and $\langle\text{head}\rangle$ is a sequence of one field access to reach the affected memory location. We sometimes refer to locations abstractly in terms of a base type rather than a base value. Here, lst has type pvl_list, so the abstract access path for this field access is $(\text{pvl\_list}, \langle\text{head}\rangle)$.

Following Matosevic and Abdelrahman, we construct symbolic access paths by traversing the call graph bottom-up, with strongly-connected components being iteratively re-analyzed until a fixed-point is reached. We will assume that functions are normalized such that the return value is the first parameter in the list of formal parameters (always indexed as zero). Functions return values by writing to their return parameter. Void functions have a placeholder in argument zero. For each library analyzed, we construct two maps, each keyed by function and formal parameter number:

- Let f be a function and i be the zero-based index of a parameter to that function. Then $finalizePaths[f, i]$ is a set of access paths that function f finalizes in its i^{th} parameter.
- Let f be a function and i be the zero-based index of a parameter to that function. Then $writePaths[f, i]$ is a set of triples of the form (p, j, q) where function f reads the value at access path q of its j^{th} parameter and ultimately stores this value into access path p of its i^{th} parameter.

Let $argno(v)$ return the index of formal parameter v in the formal parameter list of the enclosing function. Let $base(p)$ return the base of access path p and $components(p)$ return the path components of p. The access path extend operation $p_1 \oplus p_2$ extends p_1 by p_2 in the natural way; the resulting path has the same base value as p_1 and the path components of p_2 appended to those of p_1. The set-valued operation $nr(p)$ returns the singleton set containing p if each path component in $components(p)$ is unique within p; otherwise, it returns the empty set. Likewise, $nr((p, j, q))$ returns a singleton set containing a triple if neither p nor q has repeated path components. This condition excludes cyclic paths that could grow indefinitely; such paths are common in the presence of inductive data structures.

Our handling and representation of cyclic access paths differs from Matosevic and Abdelrahman [11]. They represent paths using a regular expression-like language with repetition operators. Our analysis does not require information about cycles in paths, so we use the simpler representation discussed above; this requires the no-repetition condition (through the $nr()$ operator) to prevent cyclic paths from growing without bound. This less expressive treatment of paths is a potential source of unsoundness, though it has not been a problem in practice.

We analyze code represented as LLVM bitcode: three-address code in static single assignment (SSA) form [8]. Two types of statements add elements to *finalizePaths* or *writePaths*: function calls and store instructions. This analysis is flow-insensitive and paths are created or extended for any relevant store or function call that *may* be executed. We consider function calls first. Suppose function f contains a function call of the form g(value). Let $p = ap(value)$ be the access path of *value*. If g is a finalizer and $base(p)$ is among the formal parameters of f, then:

$$finalizePaths[f, argno(base(p))] \cup= nr(p)$$

Now consider a store of the form *location = value in function f. Let $lp = ap(location)$ and $p = ap(value)$. If both $base(lp)$ and $base(p)$ are among the formal parameters of f, then:

$$writePaths[f, argno(base(lp))] \cup= nr((lp, argno(base(p)), p))$$

Calls to non-finalizer functions generate new paths by extending access paths in *finalizePaths* and *writePaths*. At a high level, paths are extended by mapping access paths in callees to the arguments of their callers. For each call callee(..., a, ...) in function f where a is the i^{th} argument to *callee* and $p \in finalizePaths[callee, i]$, let $pext = ap(a) \oplus p$. If $base(pext)$ is a formal parameter of f, then:

$$finalizePaths[f, argno(base(pext))] \cup= nr(pext)$$

For each call callee(..., a, ..., b, ...) in function f where a and b are the i^{th} and j^{th} arguments to *callee*, respectively, and $(p, j, q) \in writePaths[callee, i]$, let $qext = ap(b) \oplus q$ and $pext = ap(a) \oplus p$. Let $qextB = base(qext)$ and $pextB = base(pext)$. If both $qextB$ and $pextB$ are formal parameters of f, then:

$$writePaths[f, argno(pextB) \cup= nr((pext, argno(qextB), qext))$$

Lastly, assume a function f calls v = g(..., a, ...); and then later calls h(..., v, ...); where h finalizes v and a is the j^{th} argument to g. Further assume that $(p, j, q) \in writePaths[g, 0]$ and that $base(ap(a))$ is a formal parameter of f with index i. p is a degenerate access path with no components because it is the return value of g. Let $qext = ap(a) \oplus q$, the path of $base(ap(a))$ that is finalized by h. Then:

$$finalizePaths[f, i] \cup= nr(qext)$$

In each of these cases, we use local points-to information (the PT-relation from Matosevic and Abdelrahman [11]) to produce maximal access paths. For the pvl_push function in figure 2, the analysis can only conclude that d is written to the path (e, ⟨data⟩) without local points-to information. Since e is not a formal parameter of pvl_push, this fact is not recorded in *writePaths*. With local points-to information, the analysis can construct the maximal path (lst, ⟨head,data⟩). The base of this maximal path is lst, which is a formal parameter of pvl_push. Thus, *writePaths* can be updated to reflect the write of d to this path.

4.2 Identifying Owned Fields

After all of the symbolic access paths in a library are constructed, we next identify the owned fields in the library. According to our resource ownership model from section 2, owned fields of a type are those fields that will be finalized when an object of that type is finalized. Our analysis determines this by analyzing the finalized access paths of each function: if a field of the argument of a finalizer function is finalized, that field is owned. That is, if some function f is a finalizer for its i^{th} formal parameter, all of the fields in $\mathit{finalizePaths}[f, i]$ are owned fields.

In figure 3, the function icalcomponent_free is a finalizer because the c parameter (of type icalcomponent⋆) is finalized (or NULL) on every path. The pvl_free and icalcomponent_free functions are finalizers as well (implementations not shown). We see that icalcomponent_free passes the return value of pvl_pop to a finalizer. pvl_pop returns the data from the head of its list argument through the access path (lst, ⟨head,data⟩). Thus, the value passed to the finalizer is (icalcomponent, ⟨components,head,data⟩); we conclude that this field of icalcomponent is owned.

4.3 Transferred Ownership

The key insight of the ownership transfer analysis is that ownership of any function argument stored into an owned field is transferred from the caller. Assume a function f has formal parameters a and b at positions i and j in the formal parameter list respectively. If $(p, j, q) \in \mathit{writePaths}[f, i]$ where $ap(b)$ is q (q is the degenerate access path of just the formal parameter b) and the last component of p is an owned field, as per section 4.2, then f transfers ownership of b to a.

Returning to figure 3, the function pvl_push stores its d argument into a field of lst, which is summarized by the access path (lst, ⟨head,data⟩). icalcomponent_add_component calls pvl_push with a field of c, components, as an argument, extending the write access path to (c, ⟨components,head,data⟩). The first phase of the analysis identified the last component of this path as an owned field, and thus the c argument of icalcomponent_add_component assumes ownership of child. Note that icalcomponent_set_parent does not assume ownership of its parent parameter: the corresponding field is never finalized, and is thus not owned.

Our access path construction proceeds backwards from an address across pointer dereferences, field accesses, union accesses, and array accesses. We stop the construction at SSA ϕ-nodes to avoid a potential exponential explosion of generated access paths. While this is unsound, we have not observed any missed ownership transfers in practice.

5. An Improved Escape Analysis

While the results of the transfer analysis are essential to minimize the effort of generating language bindings that automate resource management, an escape analysis still provides important information. If a pointer to an object escapes, but ownership of that object is not transferred, we still learn information about the lifetime of that object. While we cannot always automatically manage this lifetime, the user can at least be informed that some scope management is required. Helper functions could be used to pin objects with scoped lifetimes to keep them from being collected while references exist in a C library that are not visible to the garbage collector. While prior work has described an escape analysis for this purpose, we present a more precise analysis that eliminates many of the false positives of prior work.

We describe a bottom-up summary-based flow-insensitive escape analysis with limited field and context sensitivity for C. This analysis is a form of stack escape analysis [3, 21], rather than a thread escape analysis [19]. Our analysis is based on a *value flow escape graph*, which is a value flow graph [9] with extra annotations. Our analysis is most similar to that of Whaley and Rinard [21], though ours is flow-insensitive and requires only a single graph per function. Like Whaley and Rinard [21], we analyze functions independently of their callers. We require this because callers may be written in another language and unavailable at analysis time. Furthermore, we trade precision for speed and simpler handling of callees. Instead of unifying the points-to escape graph of each callee into the graph of the caller at call sites, we use summary information to mark only the escaping parameters as escaping. This allows for more compact representations of callees at the cost of some of the precision of graph unification. We conservatively assume that values passed to callees with no summaries do escape. The value flow graph does not allow us to answer points-to queries, which we do not require, but it can be constructed in a single pass, unlike the points-to escape graph.

Our value flow graph has two types of nodes: *location nodes* and *sink nodes*. An edge $a \rightarrow b$ denotes that values flow from a to b. A value escapes if there is a path from it to a sink. The analysis is conservative and identifies values that *may* escape. This approximation is appropriate in that false positives (values incorrectly identified as escaping) lead to leaks, not crashes.

Sink nodes are created for (1) **return** statements in functions that return aggregate or pointer values and (2) stores to global variables, arguments, the return values of callees, and access paths thereof. A sink is also created for each escaping actual argument of a callee. The following statements induce edges in the value flow graph:

- ⋆a = b adds $b \rightarrow a$ or $b_p \rightarrow a$

  Assignments cause the source operand to flow to the destination operand. The edge $b_p \rightarrow a$ is added if the right-hand side of the assignment is a field reference with concrete access path $(b, \langle p \rangle)$; if there is no field access, the simpler form $b \rightarrow a$ is added.

- **return** a adds $a \rightarrow \mathit{sink}_{ret}$

  The returned value flows to the special return sink node.

For example, the function CaseWalkerInit in figure 5 has the value-flow escape graph shown in figure 6. We represent location nodes as circles and sinks as rectangles. The argument src is represented by its location node, which flows into a field of the w argument. src escapes because there is a path from it to a sink.

Function calls of the form f(a1, a2, ..., aN) act as a sequence of assignments $*f_1 = a1, *f_2 = a2, \ldots, *f_N = aN$ where f_i is the node representing the ith formal argument of f. If f_i allows its argument to escape, then f_i is a sink node. Local value v escapes from f if $v \rightarrow^* s$ for some $s \in \mathit{sinkNodes}$; that is if any sink node s is reachable from v. If v does not escape according to this query, then field p of v escapes from f if, $v_p \rightarrow^* s$ for some $s \in \mathit{sinkNodes}$.

To introduce a limited form of context sensitivity, we make special note of arguments that escape into fields of other arguments. Assume there is a function call f(a, b) where the first argument of f escapes into the second argument. We add an edge a → b in all callers of f to model the effects of f on the value flow escape graph of the caller. For example, the CmpIgnoreCase function in

```
void CaseWalkerInit(const char *src, CaseWalker *w) {
  w→src = src;
  w→read = 0;
}

int CmpIgnoreCase(const char *s1, const char *s2) {
  CaseWalker w1, w2;
  Char8 c1, c2;

  if (s1 == s2) return 0;

  CaseWalkerInit(s1, &w1);
  CaseWalkerInit(s2, &w2);

  for (;;) {
    c1 = CaseWalkerNext(&w1);
    c2 = CaseWalkerNext(&w2);
    if (!c1 || (c1 != c2)) break;
  }
  return (int)c1 −(int)c2;
}
```

Figure 5. Examples of escaping pointers from fontconfig library

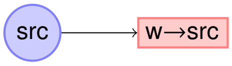

Figure 6. Value flow escape graph for CaseWalkerInit in figure 5

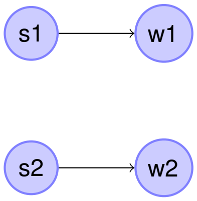

Figure 7. Value flow escape graph for CmpIgnoreCase in figure 5

figure 5 calls CaseWalkerInit, which we already established allows its src argument to escape into w. Thus, the value flow escape graph of CmpIgnoreCase, shown in figure 7, has edges from s1 to w1 and s2 and w2. Since w1 and w2 are local variables that do not otherwise escape, we are able to conclude that s1 and s2 do not escape, despite being passed as arguments that could escape if considered only in isolation.

Field sensitivity prevents a single escaping field from causing all other fields of the same object to also escape. We take a field-based approach to field sensitivity that is unsound when pointers to **struct** types are cast to structurally unrelated types [15] as in section 4.1. This unsoundness could make us label an escaping pointer as non-escaping. It could be made sound by having any casts to structurally unrelated types cause all affected fields to escape. We have not done this because such casts are rare in practice and have not yet caused problems.

6. Indirect Calls as Contracts

While analyzing library code, indirect calls (calls through function pointers) pose a problem. Libraries typically have many entry points and context of the call is unavailable. While some indirect call targets can be resolved, most cannot. Thus, whenever one of our analyses encounters an indirect call, it seems that we must conservatively assume that any function at all could be called. Instead of accepting this often overly-conservative conclusion, we note that:

```
1  typedef struct XML_ParserStruct {
2    const char *m_encoding;
3    XML_MemSuite m_mem;
4  } *XML_Parser;
5
6  XML_Parser parserCreate(const char *encodingName)
7  {
8    XML_Parser parser;
9    parser = malloc(sizeof(struct XML_ParserStruct));
10   if (parser != NULL) {
11     parser→m_encoding = encodingName;
12     parser→m_mem.malloc_fcn = malloc;
13     parser→m_mem.realloc_fcn = realloc;
14     parser→m_mem.free_fcn = free;
15   }
16
17   return parser;
18 }
19
20 DTD* dtdCreate(const XML_MemSuite *ms) {
21   DTD *p = ms→malloc_fcn(sizeof(DTD));
22   if (p == NULL) return p;
23
24   p→scaffLevel = 0;
25   p→scaffSize = 0;
26   p→scaffCount = 0;
27   p→contentStringLen = 0;
28   return p;
29 }
```

Figure 8. Contracts in expat library

- many indirect function calls are made through function pointers stored in object fields, particularly in the case where they are used to implement polymorphic behavior; and
- libraries initialize many of these fields with default functions defined in the same library.

While callers can technically store the address of any function in one of these function pointers, the library-provided initializers clearly indicate what semantics the library expects the call targets to obey. Violating those semantics is possible, but risks undefined behavior. Library-provided function pointer initializers effectively induce *contracts* on internally-initialized function pointers. The library obeys these contracts, and inferences based on them hold as long as client code obeys them as well. Recognizing these contracts and incorporating them into the analyses discussed in sections 3 to 7 improves precision.

Consider the example in figure 8. When parser objects are created in parserCreate, the malloc_fcn field of their XML_MemSuite is initialized with a pointer to the standard malloc function on line 12. While this does not mean that every malloc_fcn field will refer to malloc, it strongly suggests that the library expects the target to be an allocator. Since we consider the assignment on line 12 to create a contract on the malloc_fcn field, we conclude that dtdCreate is an allocator because of the call-under-contract on line 21.

7. Safely Sharing Ownership

Most resources in our model must be exclusively owned in order for them to be automatically managed by a high-level language run-time system. However, we support shared ownership for reference-counted resources. Moreover, ownership can be safely shared between a low-level language and a high-level language run-time system, provided the high-level language run-time safely manipu-

```
1   typedef struct {
2     int refcount;
3     Connection *connection;
4   } PendingCall;
5
6   PendingCall* pending_call_ref(PendingCall *pending) {
7     ++pending→refcount;
8     return pending;
9   }
10
11  void pending_call_unref(PendingCall *pending) {
12    −−pending→refcount;
13    if (pending→refcount) return;
14
15    Connection *c = pending→connection;
16    free(pending);
17    connection_unref(c);
18  }
```

Figure 9. Reference counting in dbus-1 library

lates the reference count. When the high-level language acquires shared ownership of a resource, it must increment its reference count. When the high-level language is ready to relinquish ownership of the resource, it must decrement the reference count instead of calling a finalizer on it. If the high-level language finalizes the resource directly, later accesses through outstanding shared references in library code could lead to a crash. Note that shared resources may safely escape into library code (c.f. section 5) because the reference counts mediate their lifetimes.

7.1 Identifying Reference Increment and Decrement Functions

We describe an analysis to identify, for a library with reference-counted resources:

- the set of types that must be reference-counted (to know when references must be managed) and
- the functions to increment and decrement references (*IncRef* and *DecRef*, respectively) for each type (to correctly manipulate the reference counts).

We begin by identifying the *DecRef* function of a resource. Fundamentally, *DecRef*:

- takes a pointer to a resource as an argument,
- decrements an integer field of the resource, and
- if the reference count becomes zero, calls a finalizer on the argument.

A common variant of *DecRef* calls the finalizer directly without decrementing the reference count if there is only a single reference to the resource. Another interesting variant, such as from gobject-2.0, has multiple reference count decrement attempts; these arise because gobject-2.0 supports finalizers that can add references to objects that are in the process of being finalized. Instead of precisely modeling every possible variant of reference counting, we employ an over-approximation that is unsound and incomplete, but nonetheless effective.

Without loss of generality, assume that *DecRef* functions take only single argument: a pointer to a resource. First, identify all *conditional finalizers* in the library. These are functions that finalize their argument on some, but not all, paths. Building on the results of the finalizer analysis described in section 3, this is a linear scan of the instructions in each function. A conditional finalizer is a function that calls at least one finalizer on its argument but is not itself a finalizer. (If it were a finalizer, it would call a finalizer on *all* paths.)

Consider the example in figure 9, which is adapted from code in the dbus-1 library simplified for exposition. Following the algorithm, we note that pending_call_unref takes a single pointer-typed argument. It passes this argument to finalizer free on line 16. However, pending_call_unref may also return early on line 13. Therefore, pending_call_unref is not itself a finalizer, but it is a conditional finalizer.

For each conditional finalizer *cf* with argument a, *cf* is a *DecRef* function with access path $(a, \langle p \rangle)$ if it decrements an integer field of a via a sequence of field accesses p. We do not require that *cf* always decrement the reference count because some variants skip it under certain circumstances; this is a potential source of imprecision. For example, the conditional finalizer pending_call_unref in figure 9 always decrements the refcount field of its pending argument on line line 12. Therefore, pending_call_unref is a *DecRef* function with access path (pending, ⟨refcount⟩).

For each *DecRef* function and its associated access path $(a, \langle p \rangle)$, the corresponding *IncRef* function is the one that takes a single argument of the same type as a, the root of the access path, and always increments the location in the resource described by p. If there is more than one *IncRef* function for a given *DecRef* function, we do not associate them and an annotation would be required to match the desired pairs. If more than one *DecRef* function could manage a given type, a consumer of the analysis results would need an annotation to prefer one. Returning to figure 9, we find that pending_call_ref always increments the refcount field of its one argument. Therefore, pending_call_ref is the *IncRef* function corresponding to pending_call_unref.

Note that we assume that libraries correctly manage reference counts internally. We make no attempt to verify this assumption, for which other analyses already exist [4].

7.2 Identifying Reference-Counted Types

So far we have identified the *IncRef* and *DecRef* functions that manipulate reference counts. We must also determine the set of types whose reference counts are managed by these functions. Clearly, this set includes the common argument type between *IncRef* and *DecRef*. Polymorphic reference counting functions, however, manage multiple types. One way to identify the set of managed types is to note that any polymorphic *DecRef* function needs some way to perform type-specific finalization when the reference count reaches zero. The types handled by these type-specific finalization functions are the types managed by the *IncRef* and *DecRef* pair.

More formally, let c be a function that has already been identified as a *DecRef* function for some type. Identify all indirect function callees in c to which c passes its argument. For each such callee f, let τ_f be the set of types to which f casts its argument. The set of types managed by c is the type of the argument of c unioned with $\bigcup_f \tau_f$.

The example in figure 10 is simplified from the open-source library glib-2.0. Our analysis recognizes g_object_unref as a *DecRef* function because of the decrements of the ref_count field on lines 6 and 9 and the call to finalizer g_type_free_instance on line 13. Next, we identify the targets of indirect calls in g_object_unref. The only indirect call appears on line 12. As initialized in g_emblem_class_init, the only known target is g_emblem_finalize. Thus, g_object_unref and g_object_ref are the *DecRef* and *IncRef* functions for GObject and GEmblem. As in this example, the indirect callees of *DecRef* often represent *finalizers* for resource-specific data members.

This algorithm assumes that *DecRef* functions operate on structural subtypes of the input value. An alternative approach is to

```
void g_object_unref(GObject ⋆object) {
  gint oref;

  oref = g_atomic_int_get(&object→ref_count);
  if (oref > 1) {
    g_atomic_int_add(&object→ref_count, −1);
  }
  else {
    oref = g_atomic_int_add(&object→ref_count, −1);

    if (oref == 1) {
      object→klass→finalize(object);
      g_type_free_instance(object);
    }
  }
}

void g_emblem_class_init(GEmblemClass ⋆klass) {
  klass→finalize = g_emblem_finalize;
}

void g_emblem_finalize(GObject ⋆object) {
  GEmblem ⋆emblem = (GEmblem⋆)object;
  g_object_unref(emblem→icon);
}
```

Figure 10. Managed types example from glib-2.0 library

directly exploit the structural subtyping relationship and consider all structural subtypes of the input to *DecRef* as being managed. We compare these two approaches in section 8.2.

7.3 Interprocedural Reference Count Manipulation

Not all functions directly increment and decrement references. Many use auxiliary functions, particularly those that rely on atomic increments and decrements. We employ a simple analysis to summarize the effects that functions have on the integer fields of their pointer-typed arguments. This analysis also tracks these effects for arguments of type **int⋆** to accommodate reference counting functions that pass the address of their reference count field instead of the object containing it.

7.4 Benefits of Automation

For users unfamiliar with a library, the inferred annotations document that reference counts must be maintained. More importantly, automatically finding the vast majority of reference-counted types decreases a potentially large annotation burden.

8. Evaluation

We have applied the analyses described in sections 3 to 6 to a suite of fifteen open-source libraries. The libraries range in size from a few functions and a few hundred lines of code up to several thousand functions with hundreds of thousands of lines of code.

We evaluate the effectiveness of the ownership transfer analysis and the reference counting analysis separately. For the most part, libraries using a reference counting discipline do not require the results of the ownership transfer analysis because ownership is explicitly shared through the reference counts. Thus, the results of the ownership transfer analysis are typically not necessary to generate library bindings for reference-counted libraries.

8.1 Transfer Analysis

This section evaluates the effectiveness of the ownership transfer analysis, with a particular focus on the reduction in manual annotation burden compared to relying solely on the results of an escape analysis. Table 1 shows the number of inferred annotations for six libraries of various sizes. The transfer analysis was designed based on archive and ical. The remaining libraries can be considered as the test set. The second column in the table notes the number of functions in each library. The "Transfer" column reports the number of function parameters that our analysis has identified as transferring ownership from the caller. The rest of the columns break down the results of our escape analysis.

Table 1. Number of inferred transfer and escape annotations

Library Analyzed		Transferred Parameters			
Name	Functions	Transfer	Contract	Indirect	Direct
archive	267	9	33	18	47
freenect	61	2	0	5	13
fuse	188	4	5	106	28
glpk	1072	0	67	54	149
gsl	3910	39	68	21	88
ical	1045	10	0	7	142

We partition escaping parameters into three categories: contract escapes, indirect escapes, and direct escapes. *Contract escapes* are parameters that escape through calls to function pointers where some targets are known, and all of those targets agree that the parameter does not escape. We refer to these parameters as contract escapes because they only escape if the contract on the function pointer they are passed to is violated, i.e., only if the assumptions of section 6 do not apply. *Indirect escapes* are parameters that are passed as arguments to calls through function pointers for which no targets can be identified. However, it is rare in practice for parameters to truly escape through function pointers because that would make the code difficult to reason about. If a consumer of these analysis results can make this assumption, the distinction can make a significant difference. For example, the fuse client library has many more indirect escapes than direct escapes. *Direct escapes* in table 1 are the remaining escaping parameters: simple escapes, such as to global variables, that are not due to calls through function pointers.

While contract escapes are clearly not expected to escape by the library, by virtue of library-provided initializers, they offer more information still. Each indirect function call that induces contract escapes imposes a contract on the function pointer that is dereferenced for that call. We can say that any function that could be pointed to by that function pointer should obey the contract that its arguments not escape. Contract escapes are most prevalent in libraries with polymorphic behavior, as can be seen in archive. In this library, polymorphism is implemented through function pointers stored in each object. These function pointer fields are initialized with functions defined in the library when objects are created, allowing us to infer the corresponding contracts as suggested in section 6. It is important to note that we do not consider an argument to an indirect function call to be a contract escape if we merely know some of the targets of the call. We only label it as a contract escape if all known call targets agree that the parameter does not escape.

Each *transfer* annotation has an accompanying *direct escape* annotation. If the results of the ownership transfer analysis, rather than the escape analysis, are used to automate resource management, then the difference between the "Direct" and "Transfer" columns in table 1 is the number of manual annotations saved by the ownership transfer analysis. Without the ownership transfer analysis, each extra escape annotation introduces a memory leak that must be plugged with a manual annotation. This difference is striking in glpk and ical: the ownership transfer analysis saves over 100 manual annotations in each. Further, glpk does not seem to ever transfer ownership. However, as discussed in section 5, the escape annotations are still useful as object lifetime documentation to the user.

Note that the transfer analysis requires an accurate view of the finalizer functions in a given library. The finalizer analysis is beyond the scope of this paper, and the one used for this evaluation was not able to identify all of the finalizers in the libraries used for this evaluation. To compensate, we manually annotated four missed finalizers in ical and two in glpk. The finalizers in archive, freenect, and fuse were automatically identified. We have not exhaustively inspected the results for glpk or gsl and some finalizers may have been missed in those libraries; if so, our ownership transfer analysis would identify *more* transferred parameters if the missed finalizers were manually annotated. We note that manually annotating finalizers is significantly easier than manually examining possible ownership transfers because finalizers tend to follow uniform naming conventions.

In the remainder of this subsection, we provide a more detailed analysis of the results for three of the libraries in our evaluation. These three libraries were chosen because they have interesting ownership transfer properties while being small enough to thoroughly evaluate by hand.

8.1.1 ical

The ical library provides a representation of calendar data. It has many functions, but only a few actually transfer ownership of objects. The primary data structures in this library form a tree; when an item is added to a tree representing some calendar event, that tree assumes ownership of the new item.

The two analyses agree that 10 parameters induce ownership transfers. The escape analysis flags over 100 extra parameters, however. Some of these non-transferred escapes can be explained by the presence of a container API that does not own its elements: adding an item to the container causes an escape but the container does not own anything except its own internal structures. Many of the remaining discrepancies arise from parent pointers. When one item is inserted as the child of another in a tree, the parent field of the newly inserted item is updated to point to its new parent element. This causes the child to escape into the parent and the parent to escape into the child. Similarly, many functions cause one or more of their parameters to escape into themselves. These self-escapes could potentially be special-cased when generating library bindings.

One function with an escape annotation but no transfer annotation was particularly interesting. This function adds an attachment to another object; however, attachments are the one resource in ical that are reference-counted. Since attachments are reference-counted, they do not have a finalizer function that our analysis could automatically identify. Adding a manual annotation to the unref function for attachments causes the analysis to correctly identify the ownership transfer in question. This suggests that it may be prudent to consider unref functions for reference-counted resources as finalizers for the purposes of the ownership transfer analysis.

8.1.2 freenect

This library provides a driver and userspace control for Kinect hardware. Our analysis infers ownership transfer of two parameters in two different functions, freenect_init and fnusb_init. Both of these inferences are incorrect. The root cause is fnusb_init, which freenect_init calls. The parameter in question is an optional libusb_context. If the caller provides a context, the library keeps a reference to it but does not assume ownership. However, if the caller does not provide a context, fnusb_init allocates its own context over which it does assume ownership. This ownership is recorded with a flag alongside the reference to the libusb_context. This violates one of the assumptions of our ownership transfer analysis: that fields are either always owned or never owned. This particular field is sometimes owned. While these inferred transfer annotations are not correct, the analysis still relieves the user of having to provide eleven annotations to compensate for the overzealous escape analysis. The transfer analysis also reduced the amount of code that must be inspected manually to two functions.

We could adapt our analysis to make special note of fields that are only sometimes finalized. Perhaps we could restrict it to fields that are finalized only based on some flag. These conditionally-owned fields could be reported in diagnostics and in generated documentation; with this extra information, users could decide on the correct ownership semantics for their case. Although adding to the manual inspection burden for users is undesirable, our analysis can show users exactly where the ambiguity arises, limiting the scope of the inspection to just unusual finalizers, rather than potentially every function in the library.

8.1.3 fuse

The fuse library is the userspace component of the Filesystem in Userspace project for some *NIX systems. This library has only a handful of ownership transfers; three of the four are transfers of a single type. In these cases, fuse_session objects assume ownership of communication channels through constructors and an explicit fuse_session_add_chan function.

The fourth ownership transferring function, fuse_session_new, is more interesting. It creates a new fuse_session with two parameters: a **void*** for arbitrary user-provided data and a **struct** with metadata. Among the metadata is an optional function to finalize the user-provided data, which the finalizer for fuse_session objects invokes if it is present.

This function exhibits an unforeseen interaction with our assumption that fields will either always be owned or never owned: the user has control over the ownership of a field, and our analysis finds evidence within the library being analyzed that the field may be owned. This case suggests that our restriction, as well as our notion of finalizers, could benefit from more nuance. In this case, we see that a field is conditionally finalized based on a value provided by the user, whereas the example in freenect made its decision to finalize or not based on a field set by the library. While the correctness of the transfer annotation in the case of this user data parameter to fuse_session_new may depend on the preferences of the library user, the transfer analysis still saves a user from having to provide at least 24 annotations to prevent leaks due to the escape analysis. Furthermore, the user need only examine the four annotations inferred by the transfer analysis. In reading the documentation on how to call fuse_session_new, the meaning of the transfer annotation on the user data parameter would become clear.

One might be tempted to generate library bindings that never transfer ownership of user data pointers to a C library. This example shows that such special treatment for even this extremely common C idiom is not completely safe. In most cases user data is not owned, but when it is the type system bears no indication one way or another.

Most of the escaping parameters whose ownership is not transferred are other user data pointers. There are also a few instances of self-escaping parameters as in ical. A third class of escapes are due to an imprecision in the escape analysis that could be fixed. The fuse library uses non-escaping heap allocations; parameters stored into these heap allocations are reported as escaping because our escape analysis does not take advantage of the fact that pointers returned by allocators like malloc are not aliased by anything. This could be fixed by treating heap-allocated locals in the same way that we treat non-escaping stack-allocated locals.

8.2 Reference Counting Analysis

Table 2 summarizes the results of our reference counting analysis. This analysis was designed based on the dbus-1, exif, gobject-2.0, and gio-2.0 libraries. For each library, we report (1) the number of functions in the library, (2) the number of functions that are

Table 2. Number of inferred reference counting annotations. "Ref/Unref" refers to the number of inferred *IncRef* and *DecRef* function pairs, rather than single functions.

Library Analyzed		Reference Counting		
Name	Functions	Allocators	Finalizers	Ref/Unref
cairo	379	2	2	4
dbus-1	804	36	8	11
exif	142	16	5	7
fontconfig	196	51	8	3
freetype	289	9	6	3
gio-2.0	1772	58	11	9
glib-2.0	1529	228	39	16
gobject-2.0	394	15	3	1
soup-2.4	530	46	6	3

```
1 void exif_mem_free(ExifMem *mem, void *d)
2 {
3   if (!mem) return;
4   if (mem→free_func) {
5     mem→free_func(d);
6     return;
7   }
8 }
```

Figure 11. Finalizer from exif library

allocators, (3) the number of functions that are finalizers, and (4) the number of *IncRef* and *DecRef* function pairs. As with our ownership transfer analysis, the reference counting analysis requires an accurate view of the allocators and finalizers present in each library. The dbus-1 library required two manual annotations, exif required four, gobject required one, and glib required seven in order for all of the allocators and finalizers to be recognized. Some of these libraries use custom memory allocators, while others have finalizers that do not quite match the notion of a finalizer that our tools use because they include extra not-NULL checks. Figure 11 shows an example. On line 4, this finalizer checks that a function pointer that it calls is not-NULL. Extending the allocator and finalizer identification analyses is beyond the scope of this work.

This analysis is useful in several ways. First, it alerts users that their library uses reference counting. Our experiments revealed reference-counted types in libraries where we did not expect them, including ical and libusb. Reference counting is not the primary resource management discipline in either library, but is still important yet not apparent from a visual scan. More importantly, even in cases where reference counting is the primary resource management discipline, our reference counting analysis identifies both polymorphic *IncRef*/*DecRef* functions and the types they operate on. For example, in dbus-1 we recognize dbus_auth_unref and dbus_auth_ref as polymorphic managers of reference counts for three related types: DBusAuthClient, DBusAuthServer, and DBusAuth.

The gio-2.0 library highlights the importance of the reference counting analysis in the presence of polymorphic reference counters. While gio-2.0 has 9 *IncRef/DecRef* pairs that are identified by the analysis, it defines a further 138 types that are managed by the g_object_unref and g_object_ref functions defined in the gobject-2.0 library. Note that the 9 types with their own reference counting functions *cannot* be managed with the generic gobject-2.0 reference counting functions, though nothing in the names of the types reveals this. Section 7.2 describes two algorithms for recognizing which types are managed by polymorphic reference counting functions. The first relies on the presence of type-specific finalizers and the second relies only on structural subtyping. The second algorithm has been more reliable in practice, identifying 23 types as managed by the gobject-2.0 reference counting functions that were missed by the first algorithm. The first algorithm misses these types because they have no type-specific resources that need to be finalized, and so do not define a finalizer. However, they are still structural subtypes of GObject, so the second algorithm recognizes them.

9. Related Work

The C/C++ leak detector of Heine and Lam [6] is very similar in spirit to our work. They present an inference algorithm based on inequality constraints to assign an owner to each object in a program. Our algorithms work on partial programs (libraries) and are formulated in terms of symbolic access paths. Our major contributions beyond their work are to (1) recognize ownership semantics for C objects with a generalized notion of allocators and finalizers (instead of C++) and (2) incorporate shared ownership via reference counting and inferred contracts on function pointers used in library code. Rayside and Mendel [17] take a more dynamic approach with ownership profiling; they report detailed hierarchical ownership information that is more precise than what we can achieve statically. Negara et al. [13] describe an inference algorithm based on liveness analysis for identifying ownership transfer semantics in message passing applications. In cases where ownership can be proved to transfer to another process via message passing, the copy of the message can be skipped and the receiving process can assume ownership of the message directly. Boyapati et al. [1] address ownership at the type level with work on ownership types. Müller and Rudich [12] extend Universe Types to support ownership transfer. Their notion of temporary aliases correspond closely to our transient references. In effect, we infer ownership types for C. Focusing specifically on memory, Wegiel and Krintz [20] discuss methods for sharing heap-allocated objects between different managed run-time environments. They do not need to establish an owner for each heap object because the run-time environments are able to cooperate and safely share objects. Since one of our run-time systems is C, which has no facilities for such object sharing, we must infer ownership.

As discussed in section 7, we do not verify the correctness of reference count handling in the libraries we analyze. Emmi et al. [4] present an analysis to perform this complex verification building on the Blast model checker. Our work is complementary in that Emmi et al. require manual specification of the set of reference-counted types, which could be automated with our analysis.

10. Conclusion

We have described an ownership model for C resources to make sharing resources between languages in polyglot programs safer. Documenting the ownership properties of even moderately large programs by hand is difficult and labor-intensive. We have described the allocator and finalizer analyses of Ravitch et al. in terms of our ownership model. We have discussed the difficulty in relying only on an escape analysis to model the ownership of objects in C libraries: many manual annotations are required to compensate for the prevalence of escaping function parameters when true ownership transfer is rare. We addressed this difficulty by describing a new ownership transfer analysis based on our ownership model. We also argue that the results of an escape analysis are still useful for understanding the semantics of libraries. We presented a scalable and composable escape analysis to further reduce the number of false positive escape annotations. We have described trade-offs for this escape analysis in context and field sensitivity that are suitable for analyzing an important class of incomplete program: library code. We have also presented an analysis to automatically identify reference-counted types and their associated reference count management functions.

Our algorithms automatically infer hundreds of annotations describing the resource ownership semantics of fifteen significant open source libraries, significantly reducing the manual annotation burden for those wishing to generate library bindings. While these inferred annotations clearly have applications to polyglot programming, they are also useful for understanding and documenting the behavior of complex C libraries.

References

[1] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In A. Aiken and G. Morrisett, editors, *POPL*, pages 213–223. ACM, 2003. ISBN 1-58113-628-5.

[2] B.-C. Cheng and W. W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In M. S. Lam, editor, *PLDI*, pages 57–69. ACM, 2000. ISBN 1-58113-199-2.

[3] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Trans. Program. Lang. Syst.*, 25(6):876–910, 2003.

[4] M. Emmi, R. Jhala, E. Kohler, and R. Majumdar. Verifying reference counting implementations. In S. Kowalewski and A. Philippou, editors, *TACAS*, volume 5505 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2009. ISBN 978-3-642-00767-5.

[5] R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., editors. *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, 2007. ACM. ISBN 978-1-59593-786-5.

[6] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *PLDI*, pages 168–181. ACM, 2003. ISBN 1-58113-662-5.

[7] U. P. Khedker, A. Sanyal, and A. Karkare. Heap reference analysis using access graphs. *ACM Trans. Program. Lang. Syst.*, 30(1), 2007.

[8] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE Computer Society, 2004. ISBN 0-7695-2102-9.

[9] L. Li, C. Cifuentes, and N. Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In T. Gyimóthy and A. Zeller, editors, *SIGSOFT FSE*, pages 343–353. ACM, 2011. ISBN 978-1-4503-0443-6.

[10] K.-K. Ma and J. S. Foster. Inferring aliasing and encapsulation properties for Java. In Gabriel et al. [5], pages 423–440. ISBN 978-1-59593-786-5.

[11] I. Matosevic and T. S. Abdelrahman. Efficient bottom-up heap analysis for symbolic path-based data access summaries. In C. Eidt, A. M. Holler, U. Srinivasan, and S. P. Amarasinghe, editors, *CGO*, pages 252–263. ACM, 2012. ISBN 978-1-4503-1206-6.

[12] P. Müller and A. Rudich. Ownership transfer in universe types. In Gabriel et al. [5], pages 461–478. ISBN 978-1-59593-786-5.

[13] S. Negara, R. K. Karmani, and G. A. Agha. Inferring ownership transfer for efficient message passing. In C. Cascaval and P.-C. Yew, editors, *PPOPP*, pages 81–90. ACM, 2011. ISBN 978-1-4503-0119-0.

[14] T. E. Oliphant. Python for scientific computing. *Computing in Science and Engineering*, 9(3):10–20, 2007.

[15] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis of c. *ACM Trans. Program. Lang. Syst.*, 30(1), 2007.

[16] T. Ravitch, S. Jackson, E. Aderhold, and B. Liblit. Automatic Generation of Library Bindings using Static Analysis. In M. Hind and A. Diwan, editors, *PLDI*, pages 352–362. ACM, 2009. ISBN 978-1-60558-392-1.

[17] D. Rayside and L. Mendel. Object ownership profiling: a technique for finding and fixing memory leaks. In R. E. K. Stirewalt, A. Egyed, and B. Fischer, editors, *ASE*, pages 194–203. ACM, 2007. ISBN 978-1-59593-882-4.

[18] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[19] A. Salcianu and M. C. Rinard. Pointer and escape analysis for multithreaded programs. In M. T. Heath and A. Lumsdaine, editors, *PPOPP*, pages 12–23. ACM, 2001. ISBN 1-58113-346-4.

[20] M. Wegiel and C. Krintz. Cross-language, Type-safe, and Transparent Object Sharing for co-Located Managed Runtimes. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *OOPSLA*, pages 223–240. ACM, 2010. ISBN 978-1-4503-0203-6.

[21] J. Whaley and M. C. Rinard. Compositional pointer and escape analysis for Java programs. In B. Hailpern, L. M. Northrop, and A. M. Berman, editors, *OOPSLA*, pages 187–206. ACM, 1999. ISBN 1-58113-238-7.