Inferred Interface Glue: Supporting Language Interoperability with Static Analysis

by

Tristan Ravitch

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2013

Date of final oral examination: 08/09/2013

The dissertation is approved by the following members of the Final Oral Committee:

Ben Liblit, Associate Professor, Computer Sciences
Tom Reps, Professor, Computer Sciences
Susan Horwitz, Professor, Computer Sciences
Somesh Jha, Professor, Computer Sciences
Timothy Tautges, Adjunct Professor, Engineering Physics

ACKNOWLEDGMENTS

I would like to thank Aditya Thakur, Tycho Andersen, Evan Driscoll, and Steve Jackson for many productive technical discussions and valuable feedback for both practice talks and paper submissions. I would also like to thank my advisor Ben Liblit for consistent encouragement and support.

Most importantly, I thank Tycho Andersen, Polina Dudnik, James Doherty, Aubrey Barnard, Erin Barnard, and Bess Berg for numerous bicycle rides and the many other things that have made Madison enjoyable.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

INFERRED INTERFACE GLUE: SUPPORTING LANGUAGE INTEROPERABILITY WITH STATIC ANALYSIS

Tristan Ravitch

Under the supervision of Professor Ben Liblit
At the University of Wisconsin–Madison

Programs written in more than one programming language are *polyglot* programs. As high-level programming languages are adopted more widely, polyglot programs become more common because they must call code written in lower-level languages to interact with existing systems. Manually calling low level code is tedious and error prone, in part due to semantic mismatches between the languages in a polyglot program. Tools exist to automatically generate library bindings (sets of wrapper functions that call low level code). However, existing tools require extensive manual annotations provided by a programmer to yield useful results. We use static analysis of C library source code to significantly reduce this manual annotation burden.

Our work infers descriptions of the interfaces that library developers intended to provide for callers, but were unable to document due to limitations of the C type system. We infer descriptions covering three important aspects of library interfaces: (1) numeric error codes returned by functions, (2) more precise types for pointer parameters that expose deeper properties about their uses, and (3) ownership semantics for objects constructed in C.

The results of these analyses can be used to generate idiomatic library bindings that make polyglot programming easier and more natural. Furthermore, the inferred interface descriptions serve as additional documentation and aid code understanding. Some of the analyses we describe are unsound and incomplete; their results are intended to be informative and helpful to someone familiar with the library being analyzed. Our evaluation shows that our results are helpful in reducing this manual annotation burden and that the unsoundness and incompleteness in our analyses are not significant problems in practice. Based on our experience, we suggest that developers can make their libraries more amenable to inclusion in polyglot programs by keeping their interfaces as abstract as possible.

Ben Liblit

ABSTRACT

Programs written in more than one programming language are *polyglot* programs. As high-level programming languages are adopted more widely, polyglot programs become more common because they must call code written in lower-level languages to interact with existing systems. Manually calling low level code is tedious and error prone, in part due to semantic mismatches between the languages in a polyglot program. Tools exist to automatically generate library bindings (sets of wrapper functions that call low level code). However, existing tools require extensive manual annotations provided by a programmer to yield useful results. We use static analysis of C library source code to significantly reduce this manual annotation burden.

Our work infers descriptions of the interfaces that library developers intended to provide for callers, but were unable to document due to limitations of the C type system. We infer descriptions covering three important aspects of library interfaces: (1) numeric error codes returned by functions, (2) more precise types for pointer parameters that expose deeper properties about their uses, and (3) ownership semantics for objects constructed in C.

The results of these analyses can be used to generate idiomatic library bindings that make polyglot programming easier and more natural. Furthermore, the inferred interface descriptions serve as additional documentation and aid code understanding. Some of the analyses we describe are unsound and incomplete; their results are intended to be informative and helpful to someone familiar with the library being analyzed. Our evaluation shows that our results are helpful in reducing this manual annotation burden and that the unsoundness and incompleteness in our analyses are not significant problems in practice. Based on our experience, we suggest that developers can make their libraries more amenable to inclusion in polyglot programs by keeping their interfaces as abstract as possible.

1 INTRODUCTION

High-level programming languages are being adopted in many application domains traditionally dominated by low-level languages such as C and C++. For example, Python has gained acceptance in the scientific computing community. Python, Javascript, Vala, and C# are significant contenders in the desktop application space, as well as user-space system-level utilities. In certain rapidly expanding application domains, such as web applications, low-level languages never had a significant presence. There are many reasons for this linguistic shift, including:

- high-level languages tend to be easier for beginners to learn;
- automatic memory management and more convenient default data structures make even experienced developers more productive; and
- high-level languages are memory-safe by default, offering advantages with respect to security and reliability.

Unfortunately, no language exists in a vacuum. Many important system-level services and functions are exposed only via libraries written in C (or similar low-level languages). This functionality often cannot be duplicated in the desired high-level language for a variety of reasons. One common reason is that high-level languages, by their very nature, do not expose many low-level details of the underlying hardware. This can render hardware-specific interactions impossible and also make some types of performance-critical code difficult, if not impossible, to implement. Code reuse is another important reason for high-level language programs to call code written in a lower-level language. Well-tested code has great value: its bugs are at least known, if not fixed. Rewriting code in another language risks introducing new and unknown bugs. In many situations, there may not be resources to port a body of code to a new language, even if such a port is desirable. Additionally, sharing a single library among several languages

promotes interoperability. Even if the specification or documentation is incorrect, if a library is self-consistent, all of the clients using it can still communicate.

As a result, high-level language programs must call functions written in lower-level languages, making them *polyglot* programs. Function calls from a high-level language to a low-level language are supported by the high-level language through a foreign function interface (FFI), which creates a wrapper function in the high-level language. A collection of these wrapper functions is a *library binding*.

As an example, consider the scientific computing community. Scientists are not generally trained systems programmers; furthermore, C is a dangerous and unproductive language in which to conduct science due to its silently unforgiving semantics. In this context, Python has gained a significant foothold in scientific computing as a glue layer between high-performance low-level components. Reusing high-performance libraries from a high-level language allows scientists to focus on their work, rather than chasing dangling pointers and memory corruption.

When only a few functions from a guest library are required, it may be convenient for a developer to use an FFI directly. However, building a complete binding for a large library is tedious, expensive, and error-prone. This is especially true when the interface of the library is evolving. Efforts to write and maintain bindings to significant libraries are often backed by teams of developers. Some examples include gtk2-perl, PyQt, PySide, gtkmm, and java-gnome: each of these is a team of developers maintaining a high-level language binding to a low-level library written by another team. These library binding developers represent an under-served programming community whose work will only become more important as polyglot development becomes more common. Additionally, if more of their work can be automated, these developers can spend more effort improving user-facing software rather than maintaining mechanical infrastructure.

It is impractical to hand-code library bindings with full function coverage for large libraries. Tools such as SWIG (Beazley, 2002) and ctypeslib (Heller, 2008) partially automate this process by parsing library headers and generating bindings

based on their contents. Several library binding teams, including PyGTK and PyQt, have created and maintain their own code generators based on C and C++ header files. Unfortunately, the interfaces generated by header-scanning systems do not take advantage of higher-level features offered by host languages without extensive manual annotations provided by a programmer. Fundamentally, they cannot do so because the only information available to them are the contents of C header files, which usually only contain type declarations and function type signatures. As a low-level programming language, C exposes few features to programmers. Consequently, C programmers must encode any higher level language features they wish to use with the limited features that C provides.

For example, while the C type system has an explicit representation of statically-sized arrays, it does not have one for arrays whose size is determined at run-time. C programmers represent these arrays as pointers to the first element of the array. At the type level, this is indistinguishable from any other pointer of the same type, such as one used to represent an output parameter. A binding-generation system based on header scanning cannot differentiate between the two cases. Without user provided manual annotations, a header scanning system must generate a simple and unhelpful wrapper function that takes a pointer. This simple wrapper foists the complexity of the interface onto the caller at every call site. The resulting library bindings are unidiomatic and force high-level language programmers to use their high-level language as if it were C.

While binding generators based on header scanning can produce idiomatic bindings with extensive manual annotations, manually annotating a small library is tedious and error-prone. It is often infeasible to fully manually annotate large or complex libraries. In this work, we describe and evaluate analyses of C library source code to recover the high-level features of library interfaces. While some of these analyses are unsound and incomplete, they are nevertheless useful; their results can be checked (and possibly corrected) by library developers and expert library users with much less effort than fully manually annotating an entire library. The checked and corrected results can be used to generate idiomatic library

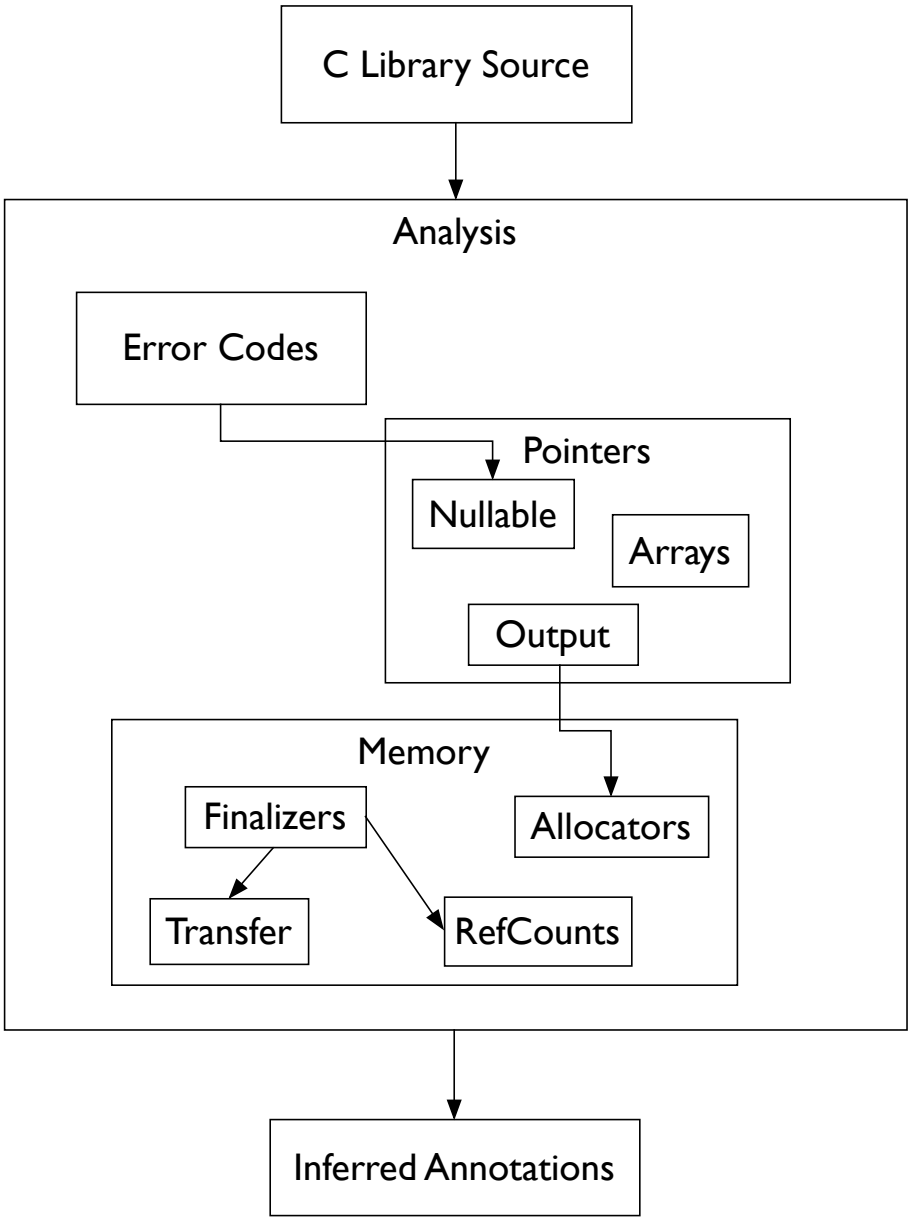

Figure 1.1: Relationships between analyses

bindings in any desired high-level programming language. The information that our analyses infer is also useful as additional API documentation, and can aid in understanding the structure and use of a library. We will discuss three general classes of analysis:

- The first analysis infers the numeric error codes that can be returned by functions (chapter 3). This information is useful to ensure that all errors have been handled by the caller. In many high-level languages, returned error codes can be converted into more idiomatic exceptions.

- Next, we describe a suite of analyses to elucidate the semantics of pointer parameters (chapter 4). Pointers are used to encode many complex constructs in C, none of which are apparent based on type signatures alone. These analyses infer deeper properties about how they are used. This information can be used to generate safer and more idiomatic bindings that automatically check invariants and handle simple but tedious conversions.

- Finally, we describe a set of analyses that infer ownership patterns for objects allocated in C libraries (chapter 5). Together, these analyses let us generate library bindings that delegate object lifetime management to the garbage collector of a high-level language. In languages with a garbage collector, this can significantly reduce the impedance mismatch between C and the high-level language. Furthermore, this information is critical even for C programmers, and it is often not documented explicitly. The results of our analyses can thus aid C programmers, even in the absence of a library binding.

The relationships between these analyses and their inputs are depicted visually in figure 1.1. Edges between analysis components indicate the direction of information flow. We conclude in chapter 6 by reviewing our results and suggesting guidelines for library developers that wish to make their libraries more amenable to polyglot programming and static analysis.

1.1 Preliminaries

This section outlines the conventions used in the rest of this document. Unless otherwise stated, the analyses in this work operate on a single function at a time, analyzing the entire call graph bottom-up. Cycles in the call graph are analyzed repeatedly until a fixed-point is reached. Indirect function call targets are resolved using Andersen's points-to analysis (Andersen, 1994). We do not assume that all call targets are visible because library clients can provide their own function pointers in most cases.

We analyze library code represented in the LLVM (version 3.3) (Lattner and Adve, 2004) IR, a three-address code in SSA form. Furthermore, the code we analyze has been subject to global value numbering (Alpern et al., 1988). Together, these transformations eliminate most local aliasing. Many of the analyses in this work do not take further measures to deal with the effects of aliasing beyond the aforementioned function pointer resolution. While our analysis has assumed access to library source code, the work of Elwazeer et al. (2013) infers data types and function signatures for binaries and produces an expressive LLVM IR output. While this binary parser is unsound and incomplete, it would provide enough information to support our analyses.

All of our analyses take as input summaries[1] of the functions in the C standard library and the functions defined by the POSIX standards. We automatically generated the summaries for chapter 3 with an analysis of the GNU C Library (GNU Project, 2013).

1.1.1 Witness Information

For each inference made by the analyses in this work, we collect witness information that consists of line numbers and reasons for the deductions. These witnesses

[1]The summaries of the standard C library are largely hand-written, but some portions are automatically generated.

are reported to the user through an annotated version of the library source. The user can examine each deduction and see why it was made.

1.1.2 Dataflow Model

Several analyses in this work are based on dataflow analysis (Kildall, 1973). In this section we briefly outline the formulation of dataflow analysis that we will use. Each analysis has:

- a dataflow fact tracked for each program point;
- an initial state that holds at the unique entry or exit point of the function being analyzed, depending on whether the analysis is forward or backward;
- a *top* element;
- a meet operator $\sqcap$; and
- a transfer function for each program point s (chosen based on the syntactic form of statement s).

1.1.3 Experimental Setup

The experiments in this document were conducted on a dual quad core Intel Xeon E5-2407 with each core clocked at 2.20 GHz and 32 gigabytes of RAM. We analyze a suite of open source libraries ranging from hundreds of lines of code to around half a million lines of code. We run all analyses on each library. The full analysis completes in less than 5 minutes for most libraries. It takes about 30 minutes for the largest library, GTK+.

2 RELATED WORK

There are two main bodies of research related to the work discussed in this document: work on automatically generating library bindings and work on checking manually-written library bindings for correctness. This section will discuss both, and their relation to our work. More specific related work will be discussed throughout the document as needed.

2.1 Library Binding Generation

The most prominent work on automatically generating foreign function interfaces is SWIG (Beazley, 2002; Beazley and Lomdahl, 1997), which scans C and C++ header files to produce library bindings. Programmers are able to provide manual annotations to improve the quality of the generated bindings. Unfortunately, the generated bindings are direct transliterations to the target high-level language of the C interface in the absence of manual annotations. Additionally, the SWIG header scanner can only handle C and C++ inputs. SWIG annotations cover some of the features and constructs discussed in this document; we could output SWIG annotations to re-use some of SWIGs code generation facilities. However, the results of some of our analyses have no SWIG analogue, and would benefit from a custom code generator.

SWIG adopted work by Reppy and Song (2006), which focused on providing data-level interoperability between host and guest languages. The work by Reppy and Song accomplishes this with *typemaps*, which specify how automatic conversions between high- and low-level data representations should be performed. The typemaps must still be generated by hand, though, placing a significant annotation burden on the user.

The work by Smolinski et al. (1999) on Babel is similar in many respects to SWIG; however, the focus is slightly different. Babel is based on an intermediate language derived from the interface description languages commonly employed

in RPC libraries: the Scientific Interface Description Language (SIDL). This intermediate language adds features useful to scientific computing, especially efficient sharing of dense numeric arrays. To fully exploit the benefits of this sharing, all of the code in a polyglot Babel program must manipulate arrays using a special array API provided by Babel. The primary focus of Babel is to generate *efficient* bindings between languages given an interface description encoded using SIDL. In contrast to SWIG, Babel supports more languages (C, C++, Java, Fortran, and Python). Furthermore, bindings can be generated to link any pair of languages without modifying the SIDL input. Unfortunately, the generation of SIDL descriptions of interfaces is entirely manual. Thus, Babel is also an interesting target output for our work. Using LLVM and gcc with the dragonegg plugin, we could analyze Fortran and Java code and re-use some of the code generation facilities of Babel.

More recently, the GNOME project introduced gobject-introspection (gobject, 2011): a sub-project to generate bindings to libraries built on the GObject framework. GObject is a C library implementing an object-oriented programming system in C that is widely used in the GNOME project. Together, GObject and gobject-introspection make a large body of code available to high-level language developers. The code generation components of gobject-introspection take library descriptions in an XML format as input. These input files describe libraries using a set of features that overlap significantly with those in this document. Unfortunately, direct comparisons between our results and the interface descriptions from gobject-introspection are not as useful as one might hope. The GNOME libraries follow a strict set of conventions that constrain C interfaces such that the results of many of our analyses are implicit in the interface. Following such a strict discipline is useful for polyglot programs that must use these libraries. In general, however, most libraries benefit from our analyses because they do not follow such strict conventions.

There are no provisions to automatically generate the interface descriptions required by gobject-introspection and Babel. Either library authors or users must

provide them. The work discussed in this document can automatically generate annotations for both projects by changing the output format from host-language code to XML interface descriptions. This could allow us to re-use the code generation facilities provided by projects like these, exploiting their specific strengths that are targeting different groups of end-users. A developer in the Babel project has expressed interest in this work, suggesting that it could have broad appeal and utility.

2.2 FFI Checking

Several bodies of work focus on checking the correctness of manually-written library bindings. Furr and Foster (2005) check the correctness of foreign calls between O'Caml and Java, with a layer of C translation code in between. They do this by implementing two type inference systems. One infers that the C code uses the O'Caml FFI correctly, while the other ensures correct usage of the Java Native Interface (JNI). Tan and Morrisett (2007) develop an inter-language analysis for Java code that approximates the effects of foreign C code on the state of the JVM. The approximated effects are presented to static analysis tools as additional JVM bytecode instructions. They extract behavioral specifications of the C code automatically using CIL. Both of these tools assume that foreign language bindings for a given library already exist and seek to verify as much as possible about their correctness statically. Not all FFI invariants are statically checkable, however, so some dynamics checks are also required. Work on Jinn by Lee et al. (2010) checks *dynamic* properties of foreign function calls. They encode the invariants that must be maintained dynamically by the foreign function interface in state machines, from which they synthesize dynamic checks.

While these techniques are effective at finding bugs in manually-written library bindings, these bugs could be avoided entirely if the bindings were automatically generated. Depending on the invariants expected by specific foreign function interfaces, it may be able to generate bindings that are correct by construction.

At the very least, all of the code interacting with the foreign function interface is consolidated in one place when bindings are automatically generated. Further, checks for the dynamic properties enforced by Jinn can be automatically and consistently inserted when bindings are automatically generated.

3 ERROR CODES

The C programming language lacks a standardized mechanism for reporting run-time errors from callees to callers. Consequently, ad-hoc approaches to reporting run-time errors to callers have proliferated. One common error reporting mechanism requires libraries to define a set of numeric error codes, either with an enumeration or as preprocessor definitions. These numeric error codes are returned as **`int`** values. Idiomatic library bindings should transform returned error codes into the native representation of the high-level language. The error code identification analysis described in this chapter will facilitate generation of library bindings that idiomatically handle returned error codes. While a seemingly simple proposition, returning error codes can take many forms:

- Return values can always be **`int`** error codes with a single distinguished code reserved to indicate success.
- Error codes could be values outside of the natural range of a function. For example, a function returning a size could return negative error codes in the case of an error, or a size otherwise.
- Functions could return a single value to signal an error, with additional details of the error available from another locations. Many POSIX functions return `-1` to signal an error, and callers must inspect `errno` for the specific error code.

This wide variety of error reporting styles can be difficult to manage correctly from C because none of the details about reported errors are encoded in the type system. Even the most well-intentioned programmer can forget to check a returned error code. The problem is exacerbated in polyglot programs because most high-level languages do not use returned **`int`** error codes to report errors. Instead, most high-level languages report errors with option types or exceptions.

As a result, programmers in these high-level languages do not expect error codes and thus do not check for them.

High-level language bindings to C libraries should transform returned `int` error codes into an idiomatic form consistent with the error reporting mechanisms of the host language. To do this, a binding generator must know 1) the set of functions in the library that can return error codes and 2) the set of error codes that the library uses. To facilitate this transformation, we propose an analysis to infer descriptions of the error reporting mechanisms of C libraries. These errors descriptions are suitable for generating library bindings. They also serve as documentation for library interfaces that cannot be expressed in the C type system. The results of this analysis are intended to aid library writers and knowledgeable library users in fully documenting library interfaces. The analysis is unsound and incomplete, but the results can dramatically reduce the manual annotation effort required to document a library interface.

At a high level, the analysis iteratively bootstraps a description of the error handling mechanism of a library that uses error codes to report errors. It starts with a description of the errors reported by the dependencies for a given library and determines how the library responds to those errors. It makes the assumption that, when a library function handles a known error, it transforms it into a new library-specific error that the function reports to its callers. Furthermore, we assume that each library has only a single style of error reporting. The former assumption is not always true. An error could be handled instead of being propagated to the caller. To account for this, the analysis also builds a model of successful computations to help refine the analysis results, eliminating the errors that this assumption can introduce. Thus, the error model is constructed from both negative and positive information. The analysis then attempts to generalize these basic facts to learn more error codes and functions that report them.

```
data ErrorAction = CalledFunction Function
                 | StoreToGlobal Int Global
                 | StoreToParameter Int Parameter

data ErrorDescriptor =
  ErrorDescriptor { errorCodes :: Set Int
                  , errorActions :: Set ErrorAction
                  }

EC :: Set Int
BD :: Map (BasicBlock, Instruction) ErrorDescriptor
```

Figure 3.1: Types used in the error code analysis

3.1 Algorithm

The analysis consists of 7 steps, which are repeated over the entire library until a fixed-point is reached:

1. Build a model of the values returned when functions *succeed*.
2. Recognize returned error codes that are transformed into new error codes.
3. Remove success codes from known errors.
4. Classify functions as error-reporting functions or normal functions.
5. Generalize based on calls to error-reporting functions.
6. Generalize based on returned integers.
7. Identify transitive error returns.

The analysis maintains two values, whose types are shown in figure 3.1 persistently across iterations. The first is *BD*: a mapping of (basic block, control predecessor) pairs to *error descriptors*. An instruction i is a control predecessor of basic block b if b is control dependent on i. The control predecessor component

allows us to distinguish between multiple errors being checked on the same branch. Each error descriptor is a record of the **int** error codes returned by the block and a set of *error actions* taken by the basic block in response to an acknowledged error. Each error action is one of:

- calling a function whose return value is not used as an argument to another function,
- storing a constant to a global variable, or
- storing a constant through a pointer parameter.

The extra condition on the first type of error action attempts to focus the analysis on functions that are called for their side effects. A function whose return value is used as an argument to another function call was not called solely for its side effects. We are interested in functions with side effects because they are often used to report warnings through logging or other mechanisms.

The second value maintained across iterations is *EC*: a set of learned **int** error codes used in the generalization steps. The error codes of each error descriptor are added to the error code set as the error descriptors are associated with basic blocks in the map. Each iteration allows the analysis to learn progressively more. The error codes returned from a function f are:

$$ErrorCodes(f) = \bigcup_{(b,c)\in P} BD[b, c].errorCodes$$

where P is the set of pairs in which:
$b \in BasicBlocks(f)$ and
$c \in ControlPredecessors(b)$.

In our model, only functions of type **int** can return error codes. Unless otherwise noted, this analysis ignores boolean functions and functions that always return the same constant **int**. **int** is often used to represent boolean values in C,

which lacks a dedicated boolean type. We heuristically identify boolean functions as those returning only the constant values zero and one.

3.1.1 Modeling Success

The analysis begins by learning models for how library functions report success. Intuitively, we identify success by looking for functions that ignore possible error codes to always return the same value on a branch. If the possibility of a call failing is being ignored, that means the library author either forgot to handle it[1] or that the failure has no effect on the result of the function. Since the return value is fixed for the branch, we assume that it must either be a success.

More precisely, we consider a branch of a function to report success if all of the following hold:

- the branch returns a constant `int` (the success code),
- the branch calls a function `f` that returns error codes,
- the result of the call to `f` is never checked for errors, and
- the call to `f` is followed by instructions with side effects (such as function calls or stores to memory).

The fourth condition is a heuristic intended to avoid identifying cleanup code on failing branches as indicating success. More stringent restrictions are possible. Figure 3.2 demonstrates why this fourth condition is necessary. The call to `cleanup_filters` on line 30 can return error codes, which are ignored. However, this code is actually an error branch that is simply cleaning up resources before signaling failure on line 31. It would be incorrect to treat `ARCHIVE_FATAL` as a success code here. The heuristic in condition four avoids this form of incorrect deduction.

[1] Note that our manually constructed interface summaries for the C standard library do not include the error returns for the `printf` family of functions because these are nearly never checked.

```
1  static int
2  build_stream(struct archive_read *a) {
3    int number_bidders, i, bid, best_bid;
4    struct archive_read_filter_bidder *bidder, *best_bidder;
5    struct archive_read_filter *filter;
6    ssize_t avail;
7    int r;
8
9    for (;;) {
10     number_bidders =
11       sizeof(a->bidders) / sizeof(a->bidders[0]);
12
13     best_bid = 0;
14     best_bidder = NULL;
15
16     bidder = a->bidders;
17     for (i = 0; i < number_bidders; i++, bidder++) {
18       if (bidder->bid != NULL) {
19         bid = (bidder->bid)(bidder, a->filter);
20         if (bid > best_bid) {
21           best_bid = bid;
22           best_bidder = bidder;
23         }
24       }
25     }
26
27     if (best_bidder == NULL) {
28       __archive_read_filter_ahead(a->filter, 1, &avail);
29       if (avail < 0) {
30         cleanup_filters(a);
31         return (ARCHIVE_FATAL);
32       }
33       a->archive.compression_name = a->filter->name;
34       a->archive.compression_code = a->filter->code;
35       return (ARCHIVE_OK);
36     }
37
38     /* ... */
39   }
40 }
```

Figure 3.2: A cleanup action whose error codes are ignored

```
int
archive_read_support_compression_program_signature(
  struct archive *_a, const char *cmd,
  const void *signature, size_t signature_len)
{
  struct archive_read *a = (struct archive_read *)_a;
  struct archive_read_filter_bidder *bidder;
  struct program_bidder *state;

  /*
   * Get a bidder object from the read core.
   */
  bidder = __archive_read_get_bidder(a);
  if (bidder == NULL)
    return (ARCHIVE_FATAL);

  /*
   * Allocate our private state.
   */
  state = (struct program_bidder*)calloc(sizeof(*state), 1);

  if (state == NULL)
    return (ARCHIVE_FATAL);
  state->cmd = strdup(cmd);
  if (signature != NULL && signature_len > 0) {
    state->signature_len = signature_len;
    state->signature = malloc(signature_len);
    memcpy(state->signature, signature, signature_len);
  }

  /*
   * Fill in the bidder object.
   */
  bidder->data = state;
  bidder->bid = program_bidder_bid;
  bidder->init = program_bidder_init;
  bidder->options = NULL;
  bidder->free = program_bidder_free;
  return (ARCHIVE_OK);
}
```

Figure 3.3: A function with success and failure codes

While instances of returned success codes meeting these criteria are not common, the instances we find, as in figure 3.3, provide useful information. In this example, we see that the possible errors returned by the call to `strdup` on line 23 are never checked. Furthermore, there are side-effecting actions (both stores and calls) between the call and the return of a constant **`int`** on line 38.

While many libraries return an **`int`** code to indicate success, not all do. There is a balance between trying to identify some success codes to refine the results of the next step in the algorithm and accidentally identifying incorrect success codes, which would hide real error codes. There is room to design more heuristics here, or possibly to apply machine learning.

3.1.2 Recognizing Transformed Error Codes

```
1 Map :: (a →b) →[a] →[b]
2
3 data Maybe a = Nothing | Just a
4
5 MapMaybe :: (a →Maybe b) →[a] →[b]
```

Figure 3.4: Overview of Option types

Next, the analysis identifies basic blocks that handle a known error condition and return a constant **`int`**. We refer to this as *error transformation*, since a known error code from a callee is transformed into a new error code. The analysis records an *error descriptor* for each basic block that transforms a known error code.

The algorithm to identify the error descriptors created by transformed error codes, which uses the definition of an option type (`Maybe`) in figure 3.4, is outlined in algorithm 1, which depends on algorithm 2. The algorithm analyzes each basic block *b* that returns a constant **`int`** in a function separately. Each control predecessor is a conditional branch predicated on a comparison. For each block, the algorithm selects a control predecessor that compares the result of a function, *callee*, that returns known error codes against a constant **`int`** on

line 4. Next, it computes the result of that call and an *error condition*, ψ, on line 5. The error condition is a formula of the form $\bigvee_{i \in e} v_{callResult} = i$ where e is the set of `int` error codes that *callee* can return. $v_{callResult}$ is a symbolic variable representing the return value of the call to *callee*. ψ encodes all of the errors that *callee* can return and is true on the branch where errors from *callee* are being checked. The algorithm also computes the facts known about the result of the call to *callee* that must hold for the scope of b.

Algorithm 1 Identifying transformed error codes

```
1:  function FunctionErrorDescriptors(f)
2:      for b ← ConstantRetBasicBlocks(f) do
3:          for cd ← ControlPredecessors(b) do
4:              if ChecksForErrors(cd) then
5:                  (callResult, ψ) ← CheckedError(cd)
6:                  case InducedFacts(b, callResult) do
7:                      Just facts :
8:                          φ ← ψ ∧ facts
9:                          if IsSatisfiable(φ) then
10:                             d ← BlockDescriptor(b)
11:                             BD[b, cd] := d
```

The formula ϕ, defined on line 8, is the key insight of the analysis. If ϕ is satisfiable, b is handling a known error code and transforming it into a new error code: the constant `int` returned by b. ϕ is satisfiable if the possible values returned by `callee` in the scope of basic block b intersect the values that `callee` returns on error. The satisfiability check on line 9 is a call to an SMT solver. If b does transform an error code, the algorithm associates the block descriptor with b in *BD* on line 11.

Algorithm 2 shows how induced facts are computed. Induced facts are the facts known about a value at the entry point of a particular basic block. The call on line 3 returns true if the control predecessor compares the call result against a constant `int`. The fact on line 4 is constructed directly from the control predecessor *dep*. The comparison has two operands (*lhs* and *rhs*) and an

Algorithm 2 Computing the facts in scope for a basic block

```
1:  function InducedFacts(b, callResult)
2:      function BuildFact(dep)
3:          if ComparesAgainstConstant(callResult, dep) then
4:              thisFact ← BasicFact(dep, callResult)
5:              case InducedFacts(dep, callResult) do
6:                  Nothing :
7:                      return (Just thisFact)
8:                  Just enclosingFacts :
9:                      return (Just (thisFact ∧ enclosingFacts))
10:         else
11:             return InducedFacts(dep, callResult)

12:     deps ← DirectControlPredecessors(b)
13:     case MapMaybe(BuildFact, deps) do
14:         [] :
15:             return Nothing
16:         fs :
17:             return (Just(⋁ fs))
```

equality or inequality relation ($\gtrless$). Without loss of generality, assume that *lhs* is *callResult*. If *rhs* is a constant `int`, we construct a formula fragment of the form $v_{callResult} \gtrless rhs$ for the control predecessor. If *dep* is on the false branch of the control predecessor branch, the formula fragment is negated.

The *MapMaybe* function is a variant of the standard *Map* function that operates over functions returning `Option` types. The type signatures for *MapMaybe* and *Maybe* are shown in figure 3.4, along with the definition of `Maybe`, an instance of an `Option` type. Intuitively, *MapMaybe* applies a function to each element of a list, collecting the returned values and discarding the *Nothing* values.

Note that this algorithm as stated only considers results that could contain errors and are compared against literals. It would be valid to check for errors against some variable. This has not been observed in practice, so we have kept this simpler formulation.

Example

The call to `calloc` in figure 3.3 on line 20 returns `NULL` if an error occurs. We apply algorithm 1 to find the error descriptors for this function. Line 22 is a basic block returning a constant. This block has two control predecessors: the first on line 14 and the second on line 21. Consider the nearest control predecessor (on line 21) first. It compares the return value of the call to `calloc` against a constant; the *callResult* is `state`, and ψ is $v_{\texttt{state}} = 0$ because `calloc` returns `NULL` on failure.

Next, we compute the induced facts using algorithm 2. There is one direct control predecessor of the block on line 22. This dependency compares the *callResult* `state` against the constant 0. The basic fact generated by this comparison is $v_{\texttt{state}} = 0$. The recursive call to *InducedFacts* returns **Nothing** because the control predecessor on line 14 does not reference `state`. Thus, *InducedFacts* returns the formula $v_{\texttt{state}} = 0$.

Returning to algorithm 1, ϕ is $v_{\texttt{state}} = 0 \wedge v_{\texttt{state}} = 0$, which is satisfiable. The error descriptor for this block contains no error actions and records the return value `ARCHIVE_FATAL`.

The reasoning is similar for the other control predecessor of this block. Starting from the control predecessor on line 14 yields a ϕ where $v_{\texttt{bidder}} = 0 \wedge v_{\texttt{bidder}} \neq 0$, which is unsatisfiable. This control predecessor does not contribute any error descriptors for this basic block.

3.1.3 Refining Error Descriptors

With models of both success and failure established, the analysis refines the set of inferred error descriptors from the error transformation analysis in section 3.1.2 with the success model from section 3.1.1. The error transformation analysis assumes that any constant returned while an error is being handled is a transformed error code. This is not always this case. This phase of the analysis considers the most frequently-occurring value among the observed success models to be the

```
1  static int
2  op_set_configuration(struct libusb_device_handle *handle,
3      int config)
4  {
5    struct linux_device_priv *priv =
6      __device_priv(handle→dev);
7    int fd = __device_handle_priv(handle)→fd;
8    int r = ioctl(fd, IOCTL_USBFS_SETCONFIG, &config);
9    if (r) {
10     if (errno == EINVAL)
11           return LIBUSB_ERROR_NOT_FOUND;
12     else if (errno == EBUSY)
13           return LIBUSB_ERROR_BUSY;
14     else if (errno == ENODEV)
15           return LIBUSB_ERROR_NO_DEVICE;
16
17     usbi_err(HANDLE_CTX(handle),
18        "failed, error %d errno %d",
19        r, errno);
20     return LIBUSB_ERROR_OTHER;
21   }
22
23   if (!sysfs_has_descriptors) {
24     /* update our cached active config descriptor */
25     if (config == -1) {
26       if (priv→config_descriptor) {
27         free(priv→config_descriptor);
28         priv→config_descriptor = NULL;
29       }
30     } else {
31       r = cache_active_config(handle→dev, fd, config);
32       if (r < 0)
33         usbi_warn(HANDLE_CTX(handle),
34                "failed to update descriptor: %d", r);
35     }
36   }
37
38   return 0;
39 }
```

Figure 3.5: An ignored error code leading to a false-positive

success code used throughout the library. Any error descriptors identified by the error transformation analysis that return the success code are discarded and the success code is removed from *EC*.

For example, the error code returned by `cache_active_config` on line 31 in figure 3.5 is handled by simply emitting a warning and then ignoring the error. However, a constant `0` is returned after this error is handled. The error transformation analysis concludes that the error code from `cache_active_config` is transformed into a constant `0`, which is true, but not indicative of error handling code. However, the success model for this library indicates that `0` is returned when functions succeed. Thus, this erroneous error descriptor can be discarded.

The analysis discovers extraneous error descriptors any time an error from a callee is handled without being propagated to the caller, or is ignored.

3.1.4 Classifying Functions to Find Error Reporters

A key component of the error descriptors collected in and refined by the previous analysis steps is the set of error actions taken while an error code is being transformed. In many libraries, these sets often include functions used to report errors by logging or by recording more detailed information about errors than is possible through returned error codes alone. Ideally, these error actions could be used to identify error handling code guarded by conditional branches that we cannot associate with known errors.

Unfortunately, these error actions contain calls to more that just error reporting functions. Code handling errors frequently calls functions to clean up resources that are no longer needed before returning an error code. Pattern matching on all of the error actions we observe in section 3.1.2 would identify any cleanup code as error reporting code. This is clearly not correct, as cleanup code occurs even more frequently outside of error handling contexts.

This step in the analysis classifies the functions appearing in sets of error actions as either error-reporting functions or other functions. Those functions

classified as error-reporting functions will be used to identify additional error codes in the next phase of the analysis. We have three classification strategies:

Null Classification

No error actions are classified as error-reporting functions. This implies that no new error codes will be learned based on error actions. This simple strategy is sufficient for some libraries.

Simple Classification

The function that is called in more than half of all error descriptors is classified as the error-reporting function for the library. This strategy is limited in that it only chooses a single function; if more than one function is used, all but one (or even all) will be missed. Furthermore, if no single function is used in more than half of the error descriptors for a library, no error-reporting function will be chosen and no new error codes will be discovered.

In the libraries we have observed that return error codes to report errors, only a single error-reporting function is used. This indicates that this simple classification heuristic may be sufficient in practice.

Machine Learning

We have also experimented with applying machine learning techniques to this classification problem. Machine learning provides many binary classification techniques that could be applied to this problem. We trained a Support Vector Machine (SVM) with LIBSVM (Chang and Lin, 2011), using the following features for each function in a set of error actions:

- the number of times the function appears in an error descriptor divided by the number of error descriptors,
- a binary feature that is `1` if the result of the function is used as the argument to another function and `0` otherwise,

- the number of times the function is called in a basic block reporting success divided by the number of basic blocks reporting success, and
- the number of times the function is called in a basic block reporting success divided by the number of times the function appears in an error descriptor.

This classifier performed poorly. It exhibited `0` percent recall, even on the training data. The culprit seems to be the skew in our training data. We have very few examples of error-reporting functions (at most one per library) and up to hundreds of other functions. Despite a clear separation in the data, there are not enough examples for the SVM algorithm to train a useful model.

The suitability of machine learning techniques to this problem is still an open question. The clear separation in the feature space between the true error-reporting functions suggests that a more nuanced training procedure may be able to overcome the skew in our label distributions. Specifically, weighting or replicating our positive examples may help.

The simple classification function only works for libraries that always use the same function to report errors. Some libraries use more than one; however, in the cases we have observed, those libraries do not return **`int`** error codes and are beyond the scope of this analysis. Approaches based on machine learning techniques should help with this more general problem. Manual examination of the feature vectors we compute indicate that machine learning may be plausible.

3.1.5 Generalizing from Calls

Using the function(s) classified as error reporters in section 3.1.4, the analysis next learns new error codes, or generalizes. If any basic block (1) does not have an error descriptor associated with it, (2) returns a constant **`int`**, and (3) calls an error-reporting function, the basic block is inferred to be error handling code.

The analysis associates an error descriptor d with the basic block and the error code is added to the set of error codes (EC):

$$\forall cd \in ControlPredecessors(b) : BD[b, cd] := d$$

The error descriptors discovered in this step allow the analysis to refine its model of the error reporting function(s) for the library.

3.1.6 Generalizing from Returns

Next, the analysis uses the set of learned `int` error codes, EC, to find new error handling blocks. Basic blocks returning a constant `int` $i \in EC$ are considered to be handling errors. For each basic block b generalized in this way, the block descriptor map is updated with a new error descriptor d:

$$\forall cd \in ControlPredecessors(b) : BD[b, cd] := d$$

These new error descriptors allow further generalization in later iterations.

3.1.7 Identifying Transitive Errors

The previous analysis phases identify returned error codes returned as constants by functions. While many error codes are returned this way, many others are returned transitively, as in figure 3.6 on lines 11 and 15. While we could handle transitive error returns trivially by merging the error descriptors from the callee and associating the result to the basic block issuing the function call, we can be more precise and filter out error codes that cannot possibly be returned. Note that this analysis differs from the others in that it operates over all functions with an `int` return type.

We can use *ErrorCodes* to define the error descriptor for a function (shown in figure 3.7). This definition is used in algorithm 3. Note that the function descriptor discards the error actions of its constituent error descriptors. In a

```
1  static int
2  header_pax_extensions(struct archive_read *a,
3      struct tar *tar,
4      struct archive_entry *entry,
5      const void *h)
6  {
7    int err, err2;
8
9    err = read_body_to_string(a, tar, &(tar→pax_header), h);
10   if (err != ARCHIVE_OK)
11     return (err);
12
13   err = tar_read_header(a, tar, entry);
14   if ((err != ARCHIVE_OK) && (err != ARCHIVE_WARN))
15     return (err);
16
17   err2 = pax_header(a, tar, entry, tar→pax_header.s);
18   err = err_combine(err, err2);
19   tar→entry_padding = 0x1ff & (-tar→entry_bytes_remaining);
20   return (err);
21 }
```

Figure 3.6: A transitive error return

transitive error return, those discarded actions are performed by the transitive callee; the caller does not perform those actions, so they are discarded.

Algorithm 3 operates on each basic block b that transitively returns error codes. For each such block, *retVal* is the value transitively returned by f. As in algorithm 1, we let ψ be a formula representing the facts we know about the program value *retVal* in basic block b. The *TransitiveReturnCallees* function returns all known callees for the call site. Each callee can return one or more return codes. Each return code *rc* is considered individually. If formula ϕ is satisfiable, then there is an assignment of values to v_{retVal} such that *rc* has not been filtered out by the conditions in scope for basic block b. This means that *rc* is a transitive return value of f and it is added to the block descriptor map *BD*.

$$
\begin{aligned}
&\mathit{FunctionDescriptor}(f) = \\
&\qquad \mathit{ErrorDescriptor}\{\mathit{errorActions} = \varnothing \\
&\qquad\qquad\qquad , \mathit{errorCodes} = \mathit{ErrorCodes}(f) \\
&\qquad\qquad\qquad \}
\end{aligned}
$$

Figure 3.7: Definition of *FunctionDescriptor*

Algorithm 3 Constructing transitive error descriptors

```
 1: function RecordTransitiveErrors(f)
 2:     for b ← TransitiveReturnBlocks(f) do
 3:         retVal ← TransitiveReturnValue(b)
 4:         ψ ← InducedFacts(b, retVal)
 5:         for callee ← TransitiveReturnCallees(b) do
 6:             for rc ← ReturnCodesFor(callee) do
 7:                 φ ← v_retVal = rc ∧ ψ
 8:                 if IsSatisfiable(φ) then
 9:                     for cd ← ControlPredecessors(b) do
10:                         d ← FunctionDescriptor(callee)
11:                         BD[b, cd] = d
```

As a concrete example, consider the transitive return on line 15. The transitively-returned value is `err`, which is the return value of *callee* `tar_read_header`. At this basic block, ψ is

$$v_{\texttt{err}} \neq \texttt{ARCHIVE\_OK} \wedge v_{\texttt{err}} \neq \texttt{ARCHIVE\_WARN}$$

For simplicity, assume that `tar_read_header` can only return two error codes: `ARCHIVE_WARN` and `ARCHIVE_FATAL`. Letting *rc* be `ARCHIVE_WARN` first, we let ϕ be

$$v_{\texttt{err}} = \texttt{ARCHIVE\_WARN} \wedge (v_{\texttt{err}} \neq \texttt{ARCHIVE\_OK} \wedge v_{\texttt{err}} \neq \texttt{ARCHIVE\_WARN})$$

This formula is clearly unsatisfiable, so ARCHIVE_WARN is not transitively returned from tar_read_header. Letting *rc* be ARCHIVE_FATAL, ϕ is

$$v_{\mathtt{err}} = \mathtt{ARCHIVE\_FATAL} \wedge (v_{\mathtt{err}} \neq \mathtt{ARCHIVE\_OK} \wedge v_{\mathtt{err}} \neq \mathtt{ARCHIVE\_WARN})$$

This formula is satisfiable, so we conclude that ARCHIVE_FATAL is transitively returned by header_pax_extensions from tar_read_header.

3.2 Related Work

Sidiroglou et al. (2009) use fuzzing to identify error handling code, which they use to implement *error virtualization*. Their fuzzing uncovers error handling code written by programmers. They then instrument programs to trap unhandled errors. The instrumentation modifies the program state to route the trapped unhandled errors to known error handling code, thus "virtualizing" the error handling code. Our work could complement their fuzzing-based approach, as a static approach may reveal more code that is difficult to reach with fuzzing tools.

The error propagation analysis of Rubio-González et al. (2009) tracks errors from the point they are generated in a program to the location at which they are handled. This flow and context sensitive interprocedural analysis for C and C++ reports dropped or otherwise unhandled errors. Their analysis requires the set of error codes to be manually specified, and they use a set of hard-coded heuristics to mark code as an error handler. Our analysis is not context sensitive, but could be complementary to the work of Rubio-González et al. in that we can automatically identify error codes.

Work on automating error recovery through compensating actions by Weimer and Necula (2004) also looks closely at error reporting and handling code. Their work, like ours, is unsound and incomplete. Unlike ours, they analyze Java and only consider checked exceptions in their analysis. Furthermore, their analysis is specialized to intraprocedural resource management errors.

Weimer and Necula (2005) mine implicit specifications for use in program verification from error handling code. Their intuition is that programs are more likely to violate specifications on infrequently executed error paths. They also expect programmers to mishandle some run-time errors, which will manifest as violations of the mined implicit specifications. Their work targets Java, and they need only identify **`catch`** blocks to find error handling code. Our work finds more general classes of error handling code that are not explicitly bracketed with a language construct.

Work by Engler et al. (2001) examines systems code written in C and is based on inferring intended library invariants based on apparent *programmer beliefs*. They give an example where calling `unlock(l)` implies that the programmer believed that `l` was locked. They use these beliefs to probabilistically infer correctness rules for systems. Their analysis is implemented as a set of code templates that they look for in code. We use a similar approach to infer error reporting functions; we treat the co-occurrence of called functions with error transforming code as significant and make further deductions based on it.

Given that error handling code is not as well tested or understood as the rest of a codebase, focusing automated testing tools (Godefroid et al., 2005) on error paths could be especially beneficial. The directed testing procedure could be directed toward the error handling paths that our analysis identifies, thus enabling more scrutiny of these poorly understood and tested portions of code.

3.3 Evaluation

To evaluate the effectiveness of the analysis presented in this chapter, we discuss several case studies of the analysis results. We have evaluated only libraries that return numeric error codes. The analysis was developed primarily around libarchive, which could be viewed as our training data. The other libraries act as test data.

```
1  /*
2   * Obsolete function provided for compatibility only.
3   * Note that the API of this function doesn't allow
4   * the caller to detect if the remaining data from the
5   * archive entry is shorter than the buffer provided,
6   * or even if an error occurred while reading data.
7   */
8  int
9  archive_read_data_into_buffer(struct archive *a,
10     void *d, ssize_t len)
11 {
12         archive_read_data(a, d, len);
13         return (ARCHIVE_OK);
14 }
```

Figure 3.8: A dangerous function in libarchive

3.3.1 Case Studies

libarchive

libarchive is a medium-sized library (about 35,000 lines of code with 267 public functions) for reading, manipulating, and writing compressed archive files in a variety of common formats. The library returns `int` error codes with `#define`d mnemonics. `0` is returned to indicate success, while error codes are mostly negative numbers. One error constant is positive and is returned to indicate that an operation reached the end of a file. Only some of the 152 functions with an `int` return type actually report errors; 72 of the `int`-typed functions cannot fail.

Our error code analysis correctly identifies the success code and recognizes 76 of the 77 functions that can return error codes. Four of the functions that return error codes call the missed function. These four functions are recognized as returning error codes, but some of the error codes that they can return are missed. As a consequence of the restrictions stated in section 3.1, the analysis misses three functions that always fail (i.e., return the same error code, and no other

```
1  static ssize_t
2  file_read(struct archive *a, void *client_data,
3      const void **buff)
4  {
5    struct read_FILE_data *mine =
6      (struct read_FILE_data *)client_data;
7    ssize_t bytes_read;
8
9    *buff = mine→buffer;
10   bytes_read = fread(mine→buffer, 1,
11       mine→block_size, mine→f);
12   if (bytes_read < 0) {
13     archive_set_error(a, errno, "Error reading file");
14   }
15   return (bytes_read);
16 }
```

Figure 3.9: A bug found in libarchive

values, on all paths) because the compile-time options disabled some features in the library:

- `archive_write_set_compression_bzip2`,
- `archive_write_set_compression_xz`, and
- `archive_write_set_compression_lzma`.

The reports generated by this analysis can help identify dangerous APIs. For example, our analysis reports that the function in figure 3.8 does not report any errors. The name suggests it should, since most read operations can fail. Indeed, the function that it wraps, `archive_read_data`, is known to return error codes. An analysis to find dropped error codes (Rubio-González et al., 2009) would report this function as suspicious. This function is actually regarded as obsolete according to the comment accompanying it in the source (included here).

However, this warning is not included in the header file that exposes this function to users.

Furthermore, this analysis revealed a bug in an internal function of libarchive. The buggy code is shown in figure 3.9. The analysis reported that `file_read` does not return any error codes. Again, as a function that reads from disk, it should be able to fail. The author of the function copied code from a similar function that called `read` and changed the low-level call to `fread`. Unfortunately, while `read` returns `-1` on error, `fread` does not. Instead, it returns a short byte count and the caller must call `ferror` on the file handle to check for an error. We reported the bug and it has since been fixed (Ravitch, 2012).

bzip2

The bzip2 library provides two APIs, one high-level and one low-level, for data compression. It is a small library with only 15 public functions. Of the public functions, 9 return **`int`**s. The analysis is unable to infer that `0` is a success code because our success code heuristic does not apply to any functions in the library. As a result, the analysis is unable to use a success model to refine the set of inferred error descriptors. The function `BZ2_bzRead` transforms all errors to `0` and reports error information to callers through an output parameter, generating error descriptors recording `0` as an error code. The analysis then incorrectly concludes that all 9 **`int`**-returning functions in bzip2 can fail with error code `0`.

Furthermore, the analysis misses error codes that are returned only in response to violated invariants, rather than lower-level system events. All 9 of the **`int`**-returning functions in bzip2 are able to return these internal-only error codes. Examples of this can be found in the function `BZ2_bzCompress` in figure 3.10. The preconditions checked on lines 4 to 7 are checking invariants that the function expects to hold for values of type `bz_stream`. Later, the function executes a state machine and returns errors if unexpected states arise, as in lines 12 and 26. More stream invariants are checked on lines 28 and 32. Since these error codes are never returned in response to system events, or even in the same functions as

```
1  int BZ_API(BZ2_bzCompress) ( bz_stream *strm, int action )
2  {
3    EState* s;
4    if (strm == NULL) return BZ_PARAM_ERROR;
5    s = strm→state;
6    if (s == NULL) return BZ_PARAM_ERROR;
7    if (s→strm != strm) return BZ_PARAM_ERROR;
8
9    preswitch:
10   switch (s→mode) {
11   case BZ_M_IDLE:
12     return BZ_SEQUENCE_ERROR;
13
14   case BZ_M_RUNNING:
15     if (action == BZ_RUN) {
16       return progress ? BZ_RUN_OK : BZ_PARAM_ERROR;
17     }
18     else if (action == BZ_FINISH) {
19       s→mode = BZ_M_FINISHING;
20       goto preswitch;
21     }
22     else
23       return BZ_PARAM_ERROR;
24
25   case BZ_M_FINISHING:
26     if (action != BZ_FINISH) return BZ_SEQUENCE_ERROR;
27     if (s→avail_in_expect != s→strm→avail_in)
28       return BZ_SEQUENCE_ERROR;
29     if (s→avail_in_expect > 0 || !isempty_RL(s) ||
30         s→state_out_pos < s→numZ) return BZ_FINISH_OK;
31     s→mode = BZ_M_IDLE;
32     return BZ_STREAM_END;
33   }
34   return BZ_OK; /*--not reached--*/
35 }
```

Figure 3.10: Internal invariant errors in bzip2

```
1  static int
2  op_kernel_driver_active(struct libusb_device_handle *handle,
3        int interface)
4  {
5    int fd = __device_handle_priv(handle)→fd;
6    struct usbfs_getdriver getdrv;
7    int r;
8
9    getdrv.interface = interface;
10   r = ioctl(fd, IOCTL_USBFS_GETDRIVER, &getdrv);
11   if (r) {
12     if (errno == ENODATA)
13       return 0;
14     else if (errno == ENODEV)
15       return LIBUSB_ERROR_NO_DEVICE;
16
17     usbi_err(HANDLE_CTX(handle),
18        "get driver failed error %d errno %d", r, errno);
19     return LIBUSB_ERROR_OTHER;
20   }
21
22   return 1;
23 }
```

Figure 3.11: A function returning a boolean or error code from libusb

system errors, the analysis is unable to learn that they are error codes.

libusb

This relatively small library (approximately 7,000 lines of code) allows user-level applications to interact with USB devices. libusb has 30 functions that return error codes. Our error code analysis correctly infers that libusb returns `0` to indicate success. The analysis correctly identifies that all 30 error reporting functions do indeed return error codes. For seven of the error reporting functions, the analysis is unable to report all possible returned error codes because four error codes are

only returned in response to purely internal errors. The analysis incorrectly infers that 1 is an error code, affecting the error descriptors of two user-facing functions.

The root cause of this incorrect inference is the `op_kernel_driver_active` function, shown in figure 3.11. This function is only used internally in the library and is not user facing, but its results propagate to users through returned values. It is a boolean function that returns true if the USB kernel driver is active. However, it can also return error codes on top of the standard true and false values. This mixing of data types (boolean values and error codes) causes the analysis to incorrectly conclude that 1 is an error code.

The analysis fails to find some error codes that are only returned in response to violated internal invariants. However, the impact on this library is less severe than it was for bzip2. All of the functions that can return error codes have a set of inferred error codes associated with them: in some cases the set is simply incomplete. Both of these errors, the extra error code and the missing error codes, would be apparent to a user familiar with the library.

sqlite3

The sqlite3 library implements a relational database in about 130,000 lines of code. This library must be robust to errors since it is trusted with a great deal of data and has many high-profile users. It uses more error codes, and more complex error codes, than the other libraries we have analyzed. While our error code analysis recognizes that sqlite3 returns `0` to indicate success and identifies many of the error codes in use in the library, it misses a significant number of error codes and functions that can return error codes. sqlite3 exports 104 functions that return **`int`** types, of which 50 cannot fail. Of the remaining 54 functions, the analysis identifies 14 functions that can return error codes, though it misses many possible error codes for each of the 14. The analysis fails to identify 30 functions that can possibly return error codes.

The problem lies in the complex error codes returned by sqlite3. An example is shown on line 25 in figure 3.12. This assignment causes the function to return

```
1  static int findCreateFileMode(const char *zPath,
2    int flags, mode_t *pMode)
3  {
4    int rc = SQLITE_OK;
5    *pMode = SQLITE_DEFAULT_FILE_PERMISSIONS;
6    if (flags & (SQLITE_OPEN_WAL|SQLITE_OPEN_MAIN_JOURNAL))
7    {
8      char zDb[MAX_PATHNAME+1];
9      int nDb;
10     struct stat sStat;
11
12     nDb = sqlite3Strlen30(zPath) - 1;
13     while ( zPath[nDb]!='-' ){
14       assert( nDb>0 );
15       assert( zPath[nDb]!='\n' );
16       nDb--;
17     }
18
19     memcpy(zDb, zPath, nDb);
20     zDb[nDb] = '\0';
21
22     if ( 0==osStat(zDb, &sStat) )
23       *pMode = sStat.st_mode & 0777;
24     else
25       rc = SQLITE_IOERR_FSTAT; // SQLITE_IOERR | (7<<8)
26
27   } else if ( flags & SQLITE_OPEN_DELETEONCLOSE ){
28     *pMode = 0600;
29   }
30   return rc;
31 }
```

Figure 3.12: A complex error code from sqlite3

```
1  int sqlite3Step(Vdbe *p)
2  {
3    /* ... */
4  end_of_step:
5    if (p→isPrepareV2 && rc!=SQLITE_ROW && rc!=SQLITE_DONE)
6    {
7      rc = sqlite3VdbeTransferError(p);
8    }
9    return (rc&db→errMask);
10 }
```

Figure 3.13: A masked error code from sqlite3

`SQLITE_IOERR_FSTAT` when the call to `osStat` fails. This is actually a complex error code that is defined as `SQLITE_IOERR | (7 << 8)`: it is a refinement of an IO error that includes extra information about the IO operation that failed.

This approach to error codes is not fundamentally a problem for our analysis. However, the additional information about errors was added late in the evolution of sqlite3 and would have broken backwards compatibility. To maintain backwards compatibility, programs calling into sqlite3 must explicitly request the more precise error codes by invoking a function provided for that purpose: `sqlite3_extended_result_codes`. Calling this function on a database connection sets a bitmask in the connection structure, which otherwise defaults to zero. All public API calls that are able to return these extended error codes actually return a masked error code, as seen in the code fragment in figure 3.13. If the caller has not opted-in to the extended error codes, the additional information is masked out.

Our analysis does not have a model for this bitmasking operation and does not recognize these masked returns as possibly returning error codes. Thus, the analysis misses all functions that can possibly return extended error codes. Furthermore, some error codes that can contain extended error codes are masked with a function call, rather than directly with a bitwise operation. Any modifications

to the analysis to handle similar constructs would need to maintain an interprocedural model of structured error codes. The sqlite3 library poses additional challenges to automated error code detection: in many functions, it temporarily stores error codes in heap-allocated structures and retrieves them farther up or down the call stack.

gsl

The GNU Scientific Library (GSL) is a numeric library providing many functions useful in scientific computing and statistics. It has nearly 4,000 functions and over 200,000 lines of code. We have not performed an exhaustive evaluation of gsl due to its size. Instead, we make a few high level observations. First, our error code analysis correctly infers that the library returns `0` (`GSL_SUCCESS`) when operations are successful. As is the case with bzip2, a number of error codes are only returned in response to violations of purely internal invariants. The error code `GSL_EDOM` is one such error code. It is returned if a function parameter is outside of the domain of the function. Our analysis never identifies it as an error code.

The results for gsl are somewhat surprising because it is primarily a math library with limited interactions with the underlying operating system. Despite this, it is able to bootstrap itself from a few error codes learned from failing IO operations and learn many new error codes throughout the rest of the library.

3.3.2 Discussion

Our error code analysis performed well on libarchive, libusb, and gsl; the results for bzip2 and sqlite3 were more mixed. The analysis had difficulty with bzip2 because that library did not have enough functions from which the analysis could generalize and recover from incorrect inferences. In particular, the inference of the library success code in section 3.1.3 is based on the relative frequencies of different success codes. With a small sample size, this can be especially imprecise.

The complex structure of the error codes in sqlite3 also cause problems. One takeaway is that the analysis is less useful for small libraries, though it did well on libusb. On the other hand, the analysis performed much better on larger libraries, which is where its results are of the most benefit to developers. While sqlite3 is a large library by line count, its user-facing API is small (as is that of bzip2). The effort required to manually annotate the error reporting semantics of these two libraries is significantly less than that of the others presented.

Error codes that are only returned in response to violations of internal invariants, and never in response to system interactions, are also a problem for this analysis. Some can be identified by the generalization steps in the algorithm, but it is clear that not all can be in real code. Identifying these error codes would be the focus of any improvements to this analysis. One approach that could work would be to generalize based on the *range* of error codes. If all of the error codes identified by the analysis are negative **`int`**s, generalizing and assuming that all negative **`int`** constants are error codes may be sensible. In a production tool, this form of generalization could be enabled by a command-line flag to the analysis tool.

A major problem in bzip2 and other libraries is that the success code is defined to be `0`: this value is useful as a success code, but is also a value that needs to be legitimately returned in contexts besides error codes. This overlap with valid return values is unfortunate and the cause of incorrect inferences in a number of libraries. It is also unnecessary, and re-defining the success constant to have no overlap with any other constants returned in the library would help the analysis avoid these errors. Doing so could also remove the need for the analysis to ignore functions heuristically identified as returning booleans. This restriction is important for our results, however, and often prevents the constant 1 as being mis-identified as an error code. The analysis reports odd resuts for boolean functions that also return error codes, as in libusb.

Finally, this analysis could be more precise if performed on an intermediate representation that preserved the lexical representation of expressions. This would

allow the analysis to more precisely distinguish between integer literals and error constants (e.g., distinguishing between the **int** 11 and the error code EAGAIN). Our analysis reports integer literals rather than the symbolic constants used in the source. Presenting reports to users in terms of the original error constant preprocessor definitions would enhance the value of the diagnostics.

4 SEMANTICS OF POINTER PARAMETERS

Pointer parameters are used for many purposes in C and C++. The notion of a pointer is very general, however, and knowing that a parameter is a pointer reveals little about how the pointer will be used. Pointers in C and C++ are used to encode many important idioms that lack explicit syntactic support in either language. For example, the function signature in figure 4.1 takes a pointer parameter. While `s` is clearly a pointer to memory, it really carries a deeper semantic meaning that is absent in the type system. Figure 4.2 shows three possible implementations with the same function signature in C. In the first, the pointer parameter `s` is used to pass an object by reference, rather than making a copy on the stack. While C++ has an explicit syntax and type for references, C does not and this idiom is common. In the second example, the pointer parameter `s` is actually an array of **`struct`**s, and the function indexes into the array. In the third example, `s` points to memory that `f` will fill in for the caller: it is an output parameter.

These type signatures admit multiple interpretations because the type systems of C and C++ are not expressive enough to encode the full semantics of pointer parameters. As a consequence, C and C++ programmers cannot determine the semantics of functions by simply examining type signatures. The problem is worse for automatic binding generation tools like SWIG and gobject-introspection, which lack the intuition that a human programmer would apply to the problem. In general, the best these tools can do without manual annotations is to create

```
1 struct S {
2   int i;
3   struct S *next;
4 };
5
6 int f(struct S *s, int val);
```

Figure 4.1: A C function signature with a pointer parameter

```
1  int f(struct S *s, int val) {
2    return s→i + val;
3  }
4
5  int f(struct S *s, int val) {
6    return s[val].i;
7  }
8
9  int f(struct S *s, int val) {
10   if (val < 0)
11     return -1;
12
13   s→i = val;
14   s→next = NULL;
15   return 0;
16 }
```

Figure 4.2: Possible implementations for figure 4.1

wrapper functions that accept pointers. It becomes the responsibility of high-level language programmers using the generated bindings to keep track of these semantic details.

We propose static program analyses to recover this deeper semantic information. The results of these analyses are inferred interface descriptions describing the uses of pointer parameters. We describe analyses to identify pointer parameters that are really:

- arrays,
- output parameters, and
- not allowed to be `NULL` (i.e., not *nullable*).

The annotations inferred by the analyses in this chapter allow library binding generators to produce safer and more idiomatic bindings. The array parameter

dimensions inferred in section 4.2 allow bindings to check to ensure that values being passed in are actually arrays. Furthermore, wrapper functions that expect array parameters can automatically convert between high-level language array representations and their C counterparts, making the bindings more idiomatic. Examples include automatically unwrapping numpy (NumPy, 2013) arrays in Python. These arrays are often used in scientific code and offer functions for C and C++ interoperability, which library bindings could invoke automatically as necessary. Similar transformations could be applied to mutable vectors in Haskell (Leshchinskiy, 2013).

Library bindings can also make output parameters more idiomatic in most languages by converting them into multiple return values (in Scheme or Common Lisp) or tuple returns (in most other languages). Programmers in these languages are accustomed to multiple return values, so the idiom would be expected. Managing the storage for output parameters automatically is also safer because it removes the possibility that a programmer misunderstands the interface and accidentally passes an invalid pointer. Since the input value of an output parameter is never used, no functionality is lost.

Identifying pointer parameters that are not nullable offers an opportunity to improve the safety of library bindings. In most languages, generated wrapper functions could add additional dynamic checks to ensure that passed pointer values are not `NULL`. These checks could throw high-level language exceptions explaining the problem, improving diagnostic power and preventing a segmentation fault in low-level code. This can stop undefined behavior before it starts. In some other languages, such as Haskell or the ML family, the story could be improved further to encode not nullable constraints in the type system through option types. With careful binding generator design, this could statically prevent a large class of errors (`NULL` pointer dereferences).

```
1  typedef struct pvl_elem_t {
2    void *data;
3    struct pvl_elem_t *next;
4  } pvl_elem;
5
6  typedef struct pvl_list_t {
7    pvl_elem *head;
8  } pvl_list;
9
10 typedef struct icalcomponent {
11   pvl_list *components;
12   struct icalcomponent* parent;
13 } icalcomponent;
14
15 void pvl_push(pvl_list *lst, void *d) {
16   pvl_elem *e = calloc(1, sizeof(pvl_elem));
17   e→next = lst→head;
18   lst→head = e;
19   e→data = d;
20 }
21
22 void* pvl_pop(pvl_list *lst) {
23   if (lst→head == NULL) return NULL;
24   list_elem *e = lst→head;
25   void *ret = e→data;
26   lst→head = e→next;
27   free(e);
28   return ret;
29 }
```

Figure 4.3: Definitions for figure 4.4

```
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 4.4: Example with access paths

4.1 Symbolic Access Paths

Our array analysis is built on *symbolic access paths* (Cheng and Hwu, 2000; Khedker et al., 2007), which we describe in this section. We follow the formulation of Matosevic and Abdelrahman (2012). An *access path* describes a memory location accessible from a base value by a (possibly empty) sequence of path components. Path components are field accesses, pointer dereferences, array accesses, and union accesses. 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 (Pearce et al., 2007), which could cause the analysis to identify invalid ownership transfers. This has not been observed in practice.

Let $ap(v)$ represent the access path for a source expression v. For example, the assignment on line 18 of figure 4.3 (taken from libical) assigns a value to `lst→head`. The corresponding access path $ap($`lst→head`$)$ is the pair (`lst`, ⟨`head`⟩). In this pair, `lst` is the base value and ⟨`head`⟩ 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 (`pvl_list`, ⟨`head`⟩).

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. For the purposes of exposition, 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 keyed by a function and a formal parameter number.

- Let f be a function and i be the zero-based index of a formal parameter to f. Then $readPaths[f, i]$ is a set of access paths that function f reads from in its i^{th} formal parameter.

- Let f be a function and i be the zero-based index of a formal parameter to f. Then $writePaths[f, i]$ is a set of access paths that function f writes to in its i^{th} formal 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. 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 (2012). 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 (enforced 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.

Three types of statements add elements to *readPaths* or *writePaths*: function calls, read instructions, 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 first consider *readPaths* and reads of the form `value = *location` in a function `f`. Recall that we analyze programs represented in an SSA-based load/store IR; the left-hand sides of store statements are analyzed before the field accesses on the right-hand side. Let $lp = ap(location)$ and $p = ap(value)$. If both $base(lp)$ and $base(p)$ are among the formal parameters of f, then:

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

For each call `callee(..., a, ...)` in function `f` where `a` is the i^{th} argument to *callee* and $p \in readPaths[callee, i]$, let $pext = ap(a) \oplus p$. If $base(pext)$ is a formal parameter of f, then:

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

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

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

For each call `callee(..., a, ...)` in function `f` where `a` is the i^{th} argument to *callee*, and $p \in writePaths[callee, i]$, let $pext = ap(a) \oplus p$. If $base(pext)$ is a formal parameter of f, then:

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

4.2 Array Parameters

This analysis identifies pointer parameters that are used as arrays. This information allows library binding generators to convert from native high-level language sequence types to C arrays. For example, where a C array is expected, generated Python bindings could accept both C arrays and numpy arrays. numpy is a popular numeric processing library for Python with support for C interoperability, and is idiomatic in many application domains. Analogous libraries in other high-level languages include the Haskell vector library, C++ vectors, and some primitive Java arrays.

Suppose a function f has a formal parameter p at zero-based index i. If $p \in readPaths[f, i]$ and every component of p is an array access, let $depth_r = length\,(components(p))$. Otherwise, $depth_r = 0$. Likewise, if $q \in writePaths[f, i]$ and every component of q is an array access, let $depth_w = length\,(components(p))$.

```
1 void glp_ios_tree_size(glp_tree *tree, int *a_cnt,
2    int *n_cnt, int *t_cnt)
3 {
4   if (a_cnt != NULL) *a_cnt = tree→a_cnt;
5   if (n_cnt != NULL) *n_cnt = tree→n_cnt;
6   if (t_cnt != NULL) *t_cnt = tree→t_cnt;
7 }
```

Figure 4.5: Conditional output parameters from GLPK

```
1 def glp_ios_tree_size(tree):
2     a_cnt = ctypes.c_int()
3     n_cnt = ctypes.c_int()
4     t_cnt = ctypes.c_int()
5     clib.glp_ios_tree_size(tree, ctypes.byref(a_cnt),
6         ctypes.byref(n_cnt), ctypes.byref(t_cnt))
7     return (a_cnt, n_cnt, t_cnt)
```

Figure 4.6: A generated wrapper for figure 4.5

Otherwise $depth_w = 0$. If either $depth_r > 0$ or $depth_w > 0$, then p is an array parameter with dimension max $(depth_r, depth_w)$.

4.3 Output Parameters

Output parameters are pointer parameters that are never read from before they are written through. They are used to return extra values (beyond the normal return value) by C and C++ functions. Note that this definition does not guarantee that output parameters are initialized on all paths. We identify output parameters so that library binding generators can create wrapper functions that convert output parameters to multiple return values in languages that support them. In most high-level programming languages, multiple return values are more idiomatic. Furthermore, allowing generated bindings to automatically manage the storage required for output parameters is less error prone than forcing the tedious details

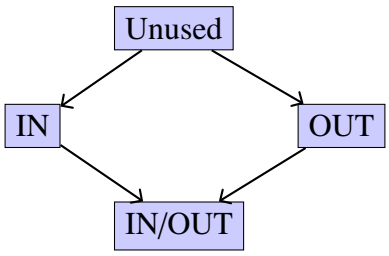

Figure 4.7: The output parameter lattice

on high-level language programmers. As an example, a library binding generator for Python might generate the function shown in figure 4.6 for the C function shown in figure 4.5. Note that this wrapper function allocates storage for three output parameters on lines 2 to 4 and then passes the addresses to the underlying C function. After the call, the wrapper returns the values returned via output parameters as a tuple to the caller in Python.

We use backward dataflow analysis to classify formal pointer parameters as elements of the bounded lattice depicted in figure 4.7. Formal parameters that are pointers to primitive types[1] are output parameters if they are classified as OUT at the entry node of their enclosing function. We analyze aggregate types (i.e., **`struct`** types) fieldwise. Pointer parameters of aggregate types are output parameters if all of their fields are classified as OUT at the entry node of their enclosing function. Both formal parameters that are pointers to primitive types and fields of formal parameters of pointer-to-aggregate type will be referred to as *input pointers*. We analyze programs represented in SSA form, so pointers can never be re-assigned.

In figure 4.7, an edge $a \rightarrow b$ means that $a \sqsupset b$ in the lattice partial order. The dataflow fact at each statement is a mapping from each input pointer to its current position in the lattice: unused (Unused), input pointer (IN), output pointer (OUT), or both (IN/OUT). The *top* element and initial fact map all input pointers to Unused. The meet operator for this analysis is pointwise greatest lower bound

[1] Primitive types are one of: **`char`**, **`short`**, **`int`**, **`long`**, **`float`**, **`double`**, and `T*` for any T.

($\sqcap$). The transfer function for statement s referencing input pointer p is defined by the following cases:

1. If s is of the form `*p = v`, the transfer function for s is:

$$\lambda outState.outState[p \mapsto \text{OUT}]$$

2. If s is of the form `v = *p`, the transfer for s is:

$$\lambda outState.outState[p \mapsto outState[p] \sqcap \text{IN}]$$

3. If s is a direct function call of the form `g(..., p, ...)` where p is an argument in positions i_0 to i_n, let d_j be the *outState* for the j^{th} formal parameter of g at g's unique entry point. Let $paramDir = \bigsqcap_{k=0\cdots n} d_k$. For each distinct argument p to g, s is analyzed as if it were a statement s' whose form is determined by the value of *paramDir*.

   - If $paramDir$ == IN, s' is `*p;`
   - If $paramDir$ == OUT, s' is `*p = arbitrary();`
   - If $paramDir$ == IN/OUT, s' is `*p = *p;`
   - Otherwise, s' is an empty statement.

4. If s is of the form `(*fp)(..., p, ...)` where `fp` is a function pointer with no known targets, the transfer function for s is:

$$\lambda outState.outState[p \mapsto \text{IN/OUT}]$$

5. Otherwise the transfer function is the identity.

Intuitively, the analysis pushes reads through each pointer parameter p towards the function entry point, stopping at writes through p. If any reads through p can

```
1  int archive_entry_xattr_next(struct archive_entry * entry,
2        const char **name, const void **value, size_t *size)
3  {
4    if (entry→xattr_p) {
5      *name = entry→xattr_p→name;
6      *value = entry→xattr_p→value;
7      *size = entry→xattr_p→size;
8
9      entry→xattr_p = entry→xattr_p→next;
10
11     return (ARCHIVE_OK);
12   } else {
13     *name = NULL;
14     *value = NULL;
15     *size = (size_t)0;
16     return (ARCHIVE_WARN);
17   }
18 }
```

Figure 4.8: An output parameter from libarchive

reach the function entry point, *p* can be read from before it is written through, making it either an IN or IN/OUT parameter. Only those pointer parameters that are classified as OUT at the function entry point are output parameters. The transfer function for pointer writes (case 1) encodes the notion that reading from an OUT parameter after it has been written cannot make it into an IN/OUT parameter. We treat calls to `memcpy` as an equivalent sequence of assignment statements if the source and destination of the copy are pointers to aggregates and the `memcpy` completely overwrites the destination. The transfer functions are all distributive (see appendix A); therefore, the analysis provides the meet over all paths solution (Kildall, 1973).

The example in figure 4.8 demonstrates a common case. This function uses its **`int`** return value to report error codes, but it still needs to return three other values: `name`, `value`, and `size`. On the first path through the function (when

entry→xattr_p is not NULL on line 4), each of these three formal parameters are written through, making them output parameters on this branch. On the other branch, they are also written through. Since OUT ⊓ OUT is OUT at the function entry, all three are output parameters of archive_entry_xattr_next.

Figure 4.9 shows an example of an aggregate output parameter. On the path where bfcp is NULL, 11 fields of the structure pointed to by parm are initialized. However, note that glp_bfcp structures have 13 fields. On this path, not all of the fields of parm are initialized. On the other hand, the path where bfcp is not NULL does initialize all of the fields of parm because the call to memcpy is treated as a sequence of field assignments (one for each field of parm). Since Unused ⊓ OUT and OUT ⊓ OUT are both OUT, all of the fields of parm are classified as OUT in the meet over all paths solution; therefore, parm is an output parameter.

This example demonstrates an important limitation of this analysis: aggregate output parameters can only be identified if *all* of their fields are initialized on some path, and none are read from before they are initialized. In the presence of reserved or unused fields, the analysis may pessimistically mis-identify them. The result in generated bindings is that callers may need to allocate storage for the aggregate manually. One strategy for handling this case would be to offer an additional manual annotation to indicate that certain fields are unused. This manual annotation would allow the analysis to handle this case more easily. Alternatively, the parameter itself may be manually annotated as an output parameter.

Another common form of output parameter is demonstrated in figure 4.5. Here, each write through a parameter is guarded by a NULL check. Since Unused⊓ OUT = OUT, all three are output parameters. These output parameters are, in effect, optional. This example highlights another interesting aspect of the output parameter analysis: automatically allocating storage for output parameters in generated library bindings prevents users from choosing the output parameters that are actually used. In an automatically generated library binding (like that shown in figure 4.6), all three output parameters (a_cnt, n_cnt, and t_cnt) are

```
typedef struct
{ /* basis factorization control parameters */
     int msg_lev; /* (reserved) */
     int type; /* factorization type: */
     int lu_size; /* luf.sv_size */
     double piv_tol; /* luf.piv_tol */
     int piv_lim; /* luf.piv_lim */
     int suhl; /* luf.suhl */
     double eps_tol; /* luf.eps_tol */
     double max_gro; /* luf.max_gro */
     int nfs_max; /* fhv.hh_max */
     double upd_tol; /* fhv.upd_tol */
     int nrs_max; /* lpf.n_max */
     int rs_size; /* lpf.v_size */
     double foo_bar[38]; /* (reserved) */
} glp_bfcp;

void glp_get_bfcp(glp_prob *lp, glp_bfcp *parm)
{
  glp_bfcp *bfcp = lp->bfcp;
  if (bfcp == NULL)
  {
    parm->type = GLP_BF_FT;
    parm->lu_size = 0;
    parm->piv_tol = 0.10;
    parm->piv_lim = 4;
    parm->suhl = GLP_ON;
    parm->eps_tol = 1e-15;
    parm->max_gro = 1e+10;
    parm->nfs_max = 50;
    parm->upd_tol = 1e-6;
    parm->nrs_max = 50;
    parm->rs_size = 0;
  }
  else
    memcpy(parm, bfcp, sizeof(glp_bfcp));
}
```

Figure 4.9: An aggregate output parameter from GLPK

always allocated. Since the generated wrapper function always passes non-`NULL` addresses to the underlying C function, the C function always initializes all three. A caller would not be able to, for example, pass `NULL` for two of the output parameters and only receive a value for the third. In figure 4.5, this is not a significant problem because the cost of allocating and initializing three **`int`**s is small. If an output parameter is expensive to initialize (e.g., it requires IO), not providing the user with a mechanism to not use these optional output parameters could be problematic.

4.3.1 Function Pointers with Known Targets

We do not currently make any inferences based on known targets of function pointers. However, they could provide some extra information. Instead of marking every parameter passed to a function pointer as IN/OUT, we could assume that known function pointers introduce a contract and take the meet over each formal parameter position of each known target as the *paramDir* for that position.

4.4 Non-Nullable Parameters

We adopt the convention that a pointer parameter `p` is *non-nullable* if passing `NULL` for `p` is guaranteed to cause an *undesirable event*. We consider the following events to be undesirable from the perspective of a caller of a library function:

- dereferencing a `NULL` pointer parameter,
- transitively calling a termination function (e.g., `exit`, `_exit`, or `abort`), or
- returning an error code (chapter 3).

A library binding generator could add dynamic checks in generated wrapper functions to prevent callers from passing `NULL` values for *non-nullable* pointer

parameters. This transformation would stop a class of errors in safe high-level language code before causing unpredictable and unrecoverable behavior in low-level C code. Library binding generators for languages with more expressive type systems could statically prevent some of the same errors, possibly using option types in Haskell and ML-derived languages[2]. A pointer parameter that the analysis fails to classify as *non-nullable* will be referred to as a *nullable* parameter.

Note that this definition will only consider a pointer parameter to be *non-nullable* if passing `NULL` for that parameter *must* cause an undesirable event. If a caller of `f` with *nullable* pointer parameter `p` passes `NULL` for `p`, `f` may still dereference a `NULL` pointer. We assume that a single path through `f` where `p` is not dereferenced implies that there is some combination of parameters and environment for which `p` is allowed to be `NULL`. We make this assumption because a definition of non-nullability preventing all `NULL` pointer dereferences could prevent callers from accessing the complete functionality of a callee. For example, the `string` parameter of function `utf8_check_string` in figure 4.10 is *nullable* according to our definition because the path that is followed for the call `utf8_check_string(NULL, 0)` does not include an undesirable event. This call is safe and useful. If we used a stronger definition of non-nullability, callers might be prevented from passing `NULL` for the `string` parameter; a generated library binding should not prevent this.

To analyze a function `f`, we rewrite calls to termination functions and statements returning error codes into a sequence of statements that dereference all of the pointer parameters of `f`. Furthermore, we rewrite calls to any function `g` with *non-nullable* parameters into dereferences of the corresponding actual arguments. Recall that we analyze the call graph bottom-up and re-analyze strongly-connected components until reaching a fixed point.

We classify pointer parameters as *non-nullable* with a forward dataflow analysis. Recall that we analyze programs represented in an SSA-based IR, so formal

[2] A *non-nullable* pointer could be represented as a type τ. Other pointers would then be type *option* τ.

```
1  int utf8_check_string(const char *string, int length)
2  {
3    int i;
4
5    if(length == -1)
6      length = strlen(string);
7
8    for(i = 0; i < length; i++) {
9      int count = utf8_check_first(string[i]);
10     if(count == 0)
11       return 0;
12     else if(count > 1) {
13       if(i + count > length)
14         return 0;
15
16       if(!utf8_check_full(&string[i], count, NULL))
17         return 0;
18
19       i += count - 1;
20     }
21   }
22
23   return 1;
24 }
```

Figure 4.10: A *nullable* pointer parameter with no check

parameters cannot be re-assigned. The dataflow fact at each program point n is the set of pointer parameters for which an undesirable event occurs on every path from the function entry to n. A pointer parameter is *non-nullable* if it is in the dataflow fact at the unique exit node of the function. The initial dataflow fact for the analysis is the empty set. The *top* element is the set of all pointer parameters in the function. The meet operation is set intersection. The transfer function for statement s is defined in two cases:

- If s dereferences pointer parameter `p`, the transfer function for s is:

$$\lambda inState.inState \cup \{p\}$$

- Otherwise, the transfer function is the identity.

Both figures 4.8 and 4.9 have formal parameters that are *non-nullable*. In figure 4.8, the `entry` parameter is dereferenced on line 4, which causes it to be added to the dataflow fact. The other three parameters are dereferenced on both branches of the conditional, which causes them to be added to the dataflow fact of each branch. When control flow merges the two branches, the sets from each branch are merged with set intersection, yielding the set containing all four formal parameters at the unique exit node. Thus, all four formal parameters are *non-nullable*.

Likewise, both of the formal parameters on line 18 of figure 4.9 are *non-nullable*. The `lp` parameter is dereferenced on line 20, which causes it to be added to the dataflow fact. On the true branch of the conditional, `parm` is also dereferenced. On the false branch of the conditional, the call to `memcpy` also makes `parm` *non-nullable*[3]. When the paths merge, the dataflow fact contains both parameters.

Of course, some pointer parameters can be *nullable*, as in figure 4.10 (already discussed) and figure 4.11. There is a path through `exif_content_get_entry` where no undesirable events occur for the `content` parameter: when `content` is `NULL`, the function simply returns `NULL`.

`NULL` pointer dereferences are not the only undesirable outcomes that can result from a pointer parameter being unexpectedly `NULL`. Figure 4.12 is an example of a library terminating the caller if a not-`NULL` invariant is violated. Passing `NULL` for `P` causes `glp_ninisat1` to terminate the program, so we want to consider it *non-nullable*. The call to `xerror` on line 5, which is transformed

[3] Annotations we provide as input to the analysis for functions from the standard C library mark the first parameter of `memcpy` as *non-nullable*.

```
ExifEntry *
exif_content_get_entry(ExifContent *content, ExifTag tag)
{
  unsigned int i;

  if (!content)
    return (NULL);

  for (i = 0; i < content→count; i++)
    if (content→entries[i]→tag == tag)
      return (content→entries[i]);
  return (NULL);
}
```

Figure 4.11: A parameter guarded with a NULL check in exif

```
int glp_minisat1(glp_prob *P)
{
  /* check problem object */
  if (P == NULL)
    xerror("glp_minisat1: P = %p; invalid object\n", P);

  /* integer solution is currently undefined */
  P→mip_stat = GLP_UNDEF;
  P→mip_obj = 0.0;

  /* ... */
}
```

Figure 4.12: A NULL check with termination from GLPK

into a dereference of `P` (as described above), is an undesirable event occurring on one path through `glp_minisat1`. On the other path through `glp_minisat1`, `P` is dereferenced. Since there are undesirable events for `P` on all paths through `glp_minisat1`, `P` is *non-nullable*.

4.4.1 Terminating Functions

This analysis considers terminating functions to be undesirable behavior. The standard C library and POSIX standard provide several functions that terminate the caller: `exit`, `_exit`, and `abort`. We transitively identify functions as terminating functions if they must call a terminating function. Furthermore, we consider a function to be a terminating function if it cannot return because it always enters an infinite loop with no body (e.g., `while(1);`). We recognize this limited class of infinite loops to support some libraries that use the idiom to implement their own `exit`-like functions.

4.5 Related Work

Our output parameter analysis is primarily motivated by the multiple return value constructs present in many languages and the approximations afforded in many others by tuples. Common Lisp (Steele, 1990, chap. 7.10.1) and Scheme (Ashley and Dybvig, 1994) define true multiple return value constructs in their language standards. Concatenative programming languages like PostScript (Horspool and Vitek, 1992) and Forth inherently support multiple return values by virtue of being able to leave as many items on the data stack as desired by the programmer. Stack effect annotations in concatenative programs formalize these multiple return values and make them apparent to programmers. Languages lacking a dedicated mechanism to return multiple values often achieve a similar effect (albeit with more memory allocation overhead) with tuples; examples of this type of language are the statically-typed functional languages (Haskell and the ML

family) and Python. C++11 provides a tuple type, but its use is not yet ubiquitous or idiomatic.

The C# language does not support multiple return values as a language construct. However, it has two parameter annotations to support output parameters: `out` and `ref` (ISO, 2006, p. 23-24). Method parameters marked with either annotation are passed by-reference. The `out` annotation is similar to our notion of OUT parameters. The exact definition is different because C# defines `out` in terms of *definite assignment*, which requires that any `out` parameter must be assigned a value along all paths by the end of the enclosing method. The `ref` annotation is likewise similar to our notion of IN/OUT parameters. The semantics of the C# `out` annotation are simpler than ours, but many uses of output parameters in C require our more liberal treatment.

Non-standard type qualifier inference (Foster et al., 1999) could be used to represent non-nullable types. That approach was adopted as a standard in C# through nullable types, which must be explicitly annotated by developers. Our notion of nullable parameters is slightly different in that we infer non-nullability based on much more broad criteria. Furthermore, our formulation is flow-sensitive.

Our work shares some ideas with the annotation language of Hackett et al. (2006), for which they also describe an inference engine. Their inferred annotations are intended for buffer overflow detection and are much more expressive in that domain. They encode `NUL` termination, read, write, preconditions, and postconditions on the states of array parameters for each function. Some of this additional information could be useful for generating library bindings. For example, buffers that are only read do not require their contents to be copied back into the high-level language caller if a sequence transformation has been applied. Our work on not-nullable pointers is also related to their `notnull` annotation. We infer `notnull` based on more broad criteria that seek to catch more than simply `NULL` pointer dereferences. In particular, we address nontermination (infinite loops), calls to termination functions, and code that reaches error paths.

4.6 Evaluation

We ran the analyses described in this chapter over a selection of open source libraries. We report only the annotations inferred for non-static functions defined in each library; the analyses infer results for more functions that are not directly callable by users. The results are shown in tables 4.1 to 4.4. Table 4.1 reports the number of functions in each library and the percent of functions in each library for which at least one annotation was inferred. The total of the "% annotated" column is the harmonic mean of the percents listed in that column. This number says that approximately 38% of functions have at least one pointer parameter whose semantics are not completely described by its type. For large libraries, this can translate to a significant number of functions and a commensurately large savings in manual annotation effort required to build idiomatic library bindings. For example, libcrypto and libgsl both expose over 3,000 functions. Our analysis infers annotations for 1,243 functions in libcrypto and 1,525 functions for libgsl. Manually providing annotations, much less manually constructing wrappers, for this many functions is impractical (as evidenced by the scarcity of complete bindings for either library). While these analyses do not save such enormous amounts of manual annotation effort on all libraries, even a few dozen automatically inferred annotations can dramatically improve the polyglot programming experience.

4.6.1 Output Parameters

The output parameter analysis described in section 4.3 infers the annotations in the "Out" and "In/Out" columns of table 4.2. Recall from the discussion of the analysis that these two pointer classifications are disjoint. A parameter can be an "Out" parameter, an "In/Out" parameter, or neither (but not both). A few trends are worth noting. First, "In/Out" parameters are not as common as "Out" parameters. Second, most libraries use "Out" parameters, which indicates that the desire to return more than one value is common. Automating the handling of

Table 4.1: Functions with Inferred Parameter Annotations

Library	**# Funcs**	**% annotated**
libacl	61	41
libarchive	267	92
libatk-1	254	6
libattr	26	31
libbz2	33	67
libcairo	379	76
libcrypto	3552	62
libcurl	334	68
libdbus-1	804	68
libexif	142	22
libexpat	80	84
libffi	23	83
libfontconfig	196	58
libfreenect	61	92
libfreetype	289	53
libfuse	188	78
libgdk-3	758	36
libgdk_pixbuf-2	121	30
libgio-2	1772	40
libglib-2	1529	72
libglpk	1072	82
libgobject-2	394	23
libgsl	3910	78
libgtk-3	4784	25
libical	1045	49
libjansson	96	45
libpango-1	406	30
libpcre	31	61
libpixman-1	128	70
libpng15	391	50
libsoup-2	530	42
libsqlite3	185	70
libssl	475	68
libusb-1	53	38
libX11	1179	75
libXau	8	62
libXdmcp	42	88
libxml2	1652	29
libXrender	45	16
libz	71	20
Total	27366	38

Table 4.2: Inferred Output Parameter Annotations

		Out		In/Out	
Library	**# Funcs**	**funcs**	**params**	**funcs**	**params**
libacl	61	8	8	3	4
libarchive	267	8	16	1	1
libatk-1	254	0	0	0	0
libattr	26	0	0	2	2
libbz2	33	9	17	2	2
libcairo	379	36	67	16	32
libcrypto	3552	564	663	506	539
libcurl	334	57	70	9	10
libdbus-1	804	88	115	46	47
libexif	142	8	10	0	0
libexpat	80	8	19	0	0
libffi	23	1	1	0	0
libfontconfig	196	27	31	4	4
libfreenect	61	15	20	0	0
libfreetype	289	67	86	14	17
libfuse	188	8	19	16	16
libgdk-3	758	63	115	11	17
libgdk_pixbuf-2	121	5	8	25	25
libgio-2	1772	76	94	262	264
libglib-2	1529	156	205	140	147
libglpk	1072	44	83	26	28
libgobject-2	394	12	13	3	3
libgsl	3910	1007	1261	44	55
libgtk-3	4784	299	463	129	149
libical	1045	72	75	8	12
libjansson	96	6	7	0	0
libpango-1	406	52	81	17	25
libpcre	31	13	19	1	2
libpixman-1	128	8	8	9	15
libpng15	391	36	96	7	12
libsoup-2	530	22	40	10	10
libsqlite3	185	42	52	9	10
libssl	475	21	27	16	18
libusb-1	53	11	11	0	0
libX11	1179	184	323	37	45
libXau	8	0	0	0	0
libXdmcp	42	9	9	0	0
libxml2	1652	51	72	16	21
libXrender	45	4	6	0	0
libz	71	3	3	3	3
Total	27366	3100	4213	1392	1535

these parameters, then, can be a significant boon to users.

We compared the results of our output parameter analysis to a hand-written binding to a subset of gsl (Jaroszewicz, 2008). Like most bindings to gsl, this binding is not complete because of its massive scope and the needs of the author. Nonetheless, our inferred annotations match the hand-coded binding very closely. We differ on only one function where the hand-written binding can return either a 0-tuple, a 1-tuple, or a 2-tuple, depending on the inputs. Our analysis considers the function to always return two values.

4.6.2 Array Parameters

Comparing the results of table 4.2 and table 4.3, many libraries have more array parameters than output parameters. This is not entirely surprising as arrays are the only collection type built in to the C language. Since C strings are actually *NUL* terminated arrays of **`char`**, the array analysis is unable to distinguish between them and other arrays of **`char`**. This is not a significant problem for many library bindings, where strings are much more common than raw byte arrays. Most binding generators could default to treating all **`char`** array parameters as strings and relying on manual annotations from a programmer to identify raw byte arrays. Our analysis is an improvement on other binding generators in this regard because we can distinguish between the case of a **`char`** used as an output parameter from **`char`** arrays.

Future work could distinguish between strings and raw byte arrays by recognizing *NUL* terminator checks, as well as seeing which pointer parameters are passed to string manipulation functions in the C standard library.

4.6.3 Non-Nullable Parameters

Table 4.4 shows that *non-nullable* pointers are very common in many libraries, more common than output parameters or array parameters. In total, more than one in three functions we analyze has a pointer parameter that is *non-nullable*.

Table 4.3: Inferred Array Parameter Annotations

		Array	
Library	**# Funcs**	**funcs**	**params**
libacl	61	8	10
libarchive	267	48	50
libatk-1	254	12	12
libattr	26	8	8
libbz2	33	5	10
libcairo	379	18	18
libcrypto	3552	482	692
libcurl	334	56	69
libdbus-1	804	92	111
libexif	142	23	23
libexpat	80	4	6
libffi	23	1	1
libfontconfig	196	60	68
libfreenect	61	4	4
libfreetype	289	10	11
libfuse	188	29	29
libgdk-3	758	28	29
libgdk_pixbuf-2	121	22	26
libgio-2	1772	317	403
libglib-2	1529	377	487
libglpk	1072	211	412
libgobject-2	394	75	127
libgsl	3910	669	890
libgtk-3	4784	335	353
libical	1045	317	324
libjansson	96	26	26
libpango-1	406	35	36
libpcre	31	15	20
libpixman-1	128	0	0
libpng15	391	64	81
libsoup-2	530	81	111
libsqlite3	185	33	34
libssl	475	27	27
libusb-1	53	2	2
libX11	1179	125	159
libXau	8	3	4
libXdmcp	42	5	13
libxml2	1652	393	536
libXrender	45	2	2
libz	71	8	8
Total	27366	4030	5232

Table 4.4: Inferred Non-Nullable Parameter Annotations

		Non-Nullable	
Library	**# Funcs**	**funcs**	**params**
libacl	61	10	12
libarchive	267	241	263
libatk-1	254	2	2
libattr	26	0	0
libbz2	33	8	9
libcairo	379	273	293
libcrypto	3552	1288	1479
libcurl	334	190	236
libdbus-1	804	446	523
libexif	142	8	8
libexpat	80	61	62
libffi	23	18	20
libfontconfig	196	82	94
libfreenect	61	54	66
libfreetype	289	95	118
libfuse	188	118	140
libgdk-3	758	203	225
libgdk_pixbuf-2	121	5	6
libgio-2	1772	227	238
libglib-2	1529	903	1026
libglpk	1072	813	956
libgobject-2	394	9	10
libgsl	3910	2417	3370
libgtk-3	4784	582	645
libical	1045	225	266
libjansson	96	26	30
libpango-1	406	60	69
libpcre	31	7	7
libpixman-1	128	80	96
libpng15	391	139	152
libsoup-2	530	174	200
libsqlite3	185	87	92
libssl	475	302	314
libusb-1	53	10	10
libX11	1179	779	976
libXau	8	4	5
libXdmcp	42	33	46
libxml2	1652	56	60
libXrender	45	4	4
libz	71	3	3
Total	27366	10042	12131

This is an important semantic feature of APIs that is not apparent from C function signatures. While we do not report specific statistics of inferred annotations on private functions in libraries, we note that `non-nullable` parameters are more common in private interfaces than in user facing interfaces. This seems to suggest that defensive programming is more common in public interfaces, while implicit invariants (such as *non-nullability*) are maintained internally.

We note an interesting interplay between the results of the output parameter analysis and the non-nullable pointer analysis. Non-nullable output parameters are required and imply that the function will always produce an extra return value. Nullable output parameters represent optional additional information that callers can request by providing storage for an output parameter. Alternatively, nullable output parameters may represent information that the function might or might not return, depending on other parameters. As noted previously, our current binding generation tools do not allow users to decide to not accept optional output parameters: we always allocate storage for them and provide no means by which to pass `NULL`. This limitation is not an inherent limitation to binding automation and merely reflects the state of our tools.

5 MEMORY OWNERSHIP

Understanding the ownership semantics and lifetimes of C objects is a fundamental prerequisite to correctly use any C library. One of our goals is to generate bindings to C libraries that automatically manage resource lifetimes; thus, these ownership semantics, which characterize correct resource flows across library boundaries, are of special interest. 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. 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 5.1. Each returns a **`char`***. The value returned by `strdup` must be `free`d to prevent memory leaks, but `free`ing 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 more sophisticated disposal than is afforded by

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

Figure 5.1: C function signatures

the generic `free` function. Consider `fopen` and `fclose`: calling `free` on the result of `fopen` is a partial resource leak. The `fclose` function is the *finalizer* for `fopen`, and is the only way to safely dispose of the result of `fopen`. Thus, functions returning managed resources must be identified along with their associated resource finalizer functions. 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 5.8, 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.

5.1 Allocators

This analysis identifies allocators, functions that allocate and return new resources to callers. While most allocators return resources to callers through their return values, others return resources through output parameters, as demonstrated in figure 5.2. We treat output parameters as additional return values and use dataflow analysis to identify the return values that hold newly-allocated values when a function returns. This analysis relies on the analysis described in section 4.3 to recognize output parameters that may be used to return allocated values. We refer to the real return value and all output parameters of a function as return slots. The return slot of the real return value is indexed by *RET*. The dataflow fact for this analysis is a mapping from return slots to sets of values returned

```
1  FT_Error
2  FT_GlyphLoader_New( FT_Memory memory,
3                      FT_GlyphLoader *aloader )
4  {
5    FT_GlyphLoader loader = NULL;
6    FT_Error error;
7
8
9    if ( !FT_NEW( loader ) )
10   {
11     loader→memory = memory;
12     *aloader = loader;
13   }
14   return error;
15 }
```

Figure 5.2: An allocation through an output parameter

through them. Here, values can be literals or function call results. Both *top* and the initial analysis fact are a mapping from each return slot to the empty set. The meet operator performs a pointwise union on each pair of return value sets. The transfer function is defined by the following cases, where s is a statement with unique successor e:

1. If s is of the form `*p = v` where `p` is an output parameter, the transfer function for edge (s, e) is

$$\lambda inState.inState[p \mapsto \{v\}]$$

2. If s is of the form **`return`** `v`, the transfer function for edge (s, e) is

$$\lambda inState.inState[RET \mapsto inState[RET] \cup \{v\}]$$

3. For all other statements, the transfer function is the identity.

```
1  struct archive_entry *archive_entry_new(void)
2  {
3    struct archive_entry *entry;
4
5    entry = (struct archive_entry *)malloc(sizeof(*entry));
6    if (entry == NULL)
7      return (NULL);
8    memset(entry, 0, sizeof(*entry));
9
10   return (entry);
11 }
```

Figure 5.3: A conventional derived allocator

Return statements have a single outgoing edge to the unique *EXIT* node of the function. A returned value or output parameter is an allocation if all of the values in its return value set at the *EXIT* node are new allocations that escape only through their respective return slot. The return value set of an allocating slot may contain `NULL` to signal an error as long as it also contains non-escaping allocations, following the convention of `malloc` established in the C standard library.

Figure 5.3 shows a simple derived allocator. At the unique exit node of `archive_entry_new`, the return value set of the real return slot contains `entry`. This value is the result of a call to `malloc`, which is a known allocator. Furthermore, since `memset` does not allow its arguments to escape, `entry` escapes only through its return slot. Thus, `archive_entry_new` is a derived allocator.

As discussed earlier in this section, the `FT_GlyphLoader_New` function in figure 5.2 is an allocator through its output parameter `aloader`. The assignment on line 12 assigns a newly allocated value to the `aloader` output parameter. At the function exit node, the return value set of `aloader` contains both `NULL` and the new allocation produced by `FT_NEW`. Thus, `FT_GlyphLoader_New` is an allocator through the `aloader` parameter.

5.2 Finalizers

We identify formal parameters that are *finalized* or `NULL` on every path through each function. While we are interested in formal parameters that are finalized, we allow finalized parameters to be `NULL` to mirror the semantics of the C `free` function. We approximate this set with a forward dataflow analysis. The dataflow fact at each program point is the set of pointer parameters that are finalized or `NULL`. The initial state of the analysis is the empty set, the *top* element is the universal set, and the meet operator is set intersection.

The transfer function for a statement s has two cases:

- If s is a call `f(..., p, ...)` where `p` is a formal parameter and is finalized by `f`, the transfer function for edge (s, e) is

$$\lambda inState.inState \cup \{p\}$$

- If s is a conditional branch statement comparing formal parameter `p` against `NULL`, the transfer function for edge (s, N) where N is the outgoing edge of s along which `p` is `NULL`

$$\lambda inState.inState \cup \{p\}$$

  Where N is the outgoing edge of s along which `p` is `NULL`.

- For all other statements, the transfer function is the identity.

If a formal parameter is an element of the dataflow fact at the unique exit node of a function, that parameter is finalized. Figure 5.4 shows two functions that finalize their formal parameters, taken from the GNU Linear Programming Toolkit (GLPK) (Makhorin, 2008). The first function, `glp_free`, is a finalizer that this analysis is unable to identify. `glp_free` calls a known finalizer on line 11, but the actual argument to this known finalizer is the result of pointer arithmetic

```
1  void glp_free(void *ptr) {
2    LIBENV *env = lib_link_env();
3    int sz = align(sizeof(LIBMEM));
4    LIBMEM *desc = (void *)((char *)ptr - sz);
5    if (desc→prev == NULL)
6      env→mem_ptr = desc→next;
7    else
8      desc→prev→next = desc→next;
9
10   env→mem_total -= desc→size;
11   free(desc);
12 }
13
14 void glp_delete_prob(glp_prob *lp) {
15   if (!lp)
16     return;
17   glp_free(lp→row);
18   glp_free(lp→col);
19   glp_free(lp);
20 }
```

Figure 5.4: Two finalizers from GLPK

```
1 {"glp_malloc" : [ [{"FAAllocator" : "glp_free"}], [] ] }
```

Figure 5.5: Manual annotations for GLPK

(line 4) that is accounting for an object header introduced by the function that originally allocated the memory. Thus, the analysis cannot determine that the `ptr` formal parameter is finalized.

While heuristics could be added to the analysis to handle object headers of this form, we do not do so because the construct is not common in practice. Recognizing `glp_free` as a finalizer requires a manual annotation as shown in figure 5.5. This manual annotation declares that `glp_free` is the function of one argument that finalizes values returned by `glp_malloc`.

With `glp_free` registered as a finalizer, the analysis can proceed to a caller of `glp_free`: `glp_delete_prob`. There are two paths through this function. On the first path (line 15), formal parameter `lp` is `NULL`. On the other path, `lp` is finalized by a call to known finalizer `glp_free` on line 19.

5.3 Symbolic Access Paths Revisited

In this section, we extend the formulation of symbolic access paths described in section 4.1 with two additional maps 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 $\mathit{finalizePaths}[f, i]$ 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 formal parameter to f. Then $\mathit{writePaths}[f, i]$ 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 $\mathit{argno}(v)$ return the index of formal parameter v in the formal parameter list of the enclosing function. Let $\mathit{base}(p)$ return the base of access path p and $\mathit{components}(p)$ 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 $\mathit{components}(p)$ 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.

Only function calls add elements to *finalizePaths*. This analysis is flow-insensitive and paths are created or extended for any relevant function call that *may* be executed. 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:

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

Calls to non-finalizer functions generate new paths by extending access paths in *finalizePaths*. 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:

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

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:

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

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:

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

For each call `callee(..., a, ..., b, ...)` c in function `f` where `a` and `b` are the i^{th} and j^{th} arguments to c, respectively, and $(p, j, q) \in writePaths[c, i]$, 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:

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

In each of these cases, we use local points-to information (the PT-relation from Matosevic and Abdelrahman (2012)) to produce maximal access paths. For the `pvl_push` function in figure 4.3, 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.

5.4 Ownership Transfer

We adopt the ownership model of Heine and Lam (2003) 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.

5.4.1 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 4.3 and 4.4. 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 in figure 4.4.

Ownership extends beyond just allocators and finalizers. For example, the function `icalcomponent_remove_component` removes a child component from a component without finalizing it. 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 (Sagiv et al., 2002).

```
1  def doFinalize(self):
2      try:
3          if self.__finalizer:
4              self.__finalizer(self)
5      except AttributeError: pass
6
7  def RPOINTER(cls, finalizer, p):
8      klass = POINTER(cls)
9      c = klass(p)
10     setattr(c, "__finalizer", finalizer)
11     setattr(c, "__del__", doFinalize)
12
13 class Foo(ctypes.Structure):
14     _fields_ = []
15
16 def new_foo():
17     c = clib.new_foo()
18     return RPOINTER(Foo, clib.free_foo, c)
19
20 # Use
21 f = new_foo()
22 clib.use_foo(f)
```

Figure 5.6: A Python object wrapper

5.4.2 Using Ownership in Generated Library Bindings

In this section, we outline how ownership properties of C libraries can be used to create library bindings that automatically manage the lifetimes of objects allocated by C functions. 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. Figure 5.6 shows one possible approach to arranging for the Python garbage collector to manage C resources. Instead of calling a C allocation function (`clib.new_foo`) directly, callers instead call the wrapper function `new_foo`. The wrapper function attaches

```
@contextmanager
def foo():
    c = clib.new_foo()
    try:
        yield c
    finally:
        clib.free_foo(c)

# Use
with foo() as f:
    clib.use_foo(f)
```

Figure 5.7: A Python deterministic finalizer

the corresponding finalizer function to the newly-allocated object. The finalizer function will be run automatically by the Python garbage collector. 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, in this case the `__del__` method, 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. As an alternative to garbage collection, Python interfaces could implement deterministic finalization through Python context managers, as shown in figure 5.7.

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 Assume that low-level language resource r is owned exclusively by a high-level language run-time and 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 owner-

```
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 5.8: Pinning Python objects with a context manager

ship 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 5.8. 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.

5.4.3 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 5.4, 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 4.4, 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.

5.4.4 Transferred Ownership of Parameters

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 $(p, j, q) \in writePaths[f, i]$ 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 5.4.3, then `f` transfers ownership of `b` to `a`.

Returning to figures 4.3 and 4.4, the function `pvl_push` stores its `d` argument into a field of `lst`, which is summarized by the access path (`lst`, ⟨`head`,`data`⟩). The `icalcomponent_add_component` function calls `pvl_push` with a field of `c`, `components`, as an argument, thereby 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.

5.5 Escape Analysis and Lifetime

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 (Choi et al., 2003; Whaley and Rinard, 1999), rather than a thread escape analysis (Salcianu and Rinard, 2001). Our analysis is based on a *value flow escape graph*, which is a value flow graph (Li et al., 2011) with extra annotations. Our analysis is most similar to that of Whaley and Rinard (1999), though ours is flow-insensitive and requires only a single graph per function. Like Whaley and Rinard (1999), 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

```
1  void CaseWalkerInit(const char *src, CaseWalker *w) {
2    w→src = src;
3    w→read = 0;
4  }
5
6  int CmpIgnoreCase(const char *s1, const char *s2) {
7    CaseWalker w1, w2;
8    Char8 c1, c2;
9
10   if (s1 == s2) return 0;
11
12   CaseWalkerInit(s1, &w1);
13   CaseWalkerInit(s2, &w2);
14
15   for (;;) {
16     c1 = CaseWalkerNext(&w1);
17     c2 = CaseWalkerNext(&w2);
18     if (!c1 || (c1 != c2)) break;
19   }
20   return (int)c1 - (int)c2;
21 }
```

Figure 5.9: Examples of escaping pointers from fontconfig library

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 $a \rightarrow sink_{ret}$

  The returned value flows to the special return sink node.

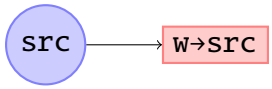

Figure 5.10: Value flow escape graph for `CaseWalkerInit` in figure 5.9

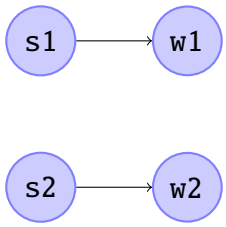

Figure 5.11: Value flow escape graph for `CmpIgnoreCase` in figure 5.9

For example, the function `CaseWalkerInit` in figure 5.9 has the value-flow escape graph shown in figure 5.10. 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 $s \in sinkNodes$; 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 $s \in sinkNodes$.

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` $\rightarrow$ `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 figure 5.9 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 5.11, 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 (Pearce et al., 2007) as in section 5.3. 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.

5.6 Shared Ownership and Reference Counting

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 manipulates 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 5.5) because the reference counts mediate their lifetimes.

```
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 5.12: Reference counting in dbus-1 library

5.6.1 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 a 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 5.2, 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 5.12, 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 5.12 always decrements the `refcount` field of its `pending` argument

on 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 5.12, 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 (Emmi et al., 2009).

5.6.2 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

```
1  void g_object_unref(GObject *object) {
2    gint oref;
3
4    oref = g_atomic_int_get(&object→ref_count);
5    if (oref > 1) {
6      g_atomic_int_add(&object→ref_count, -1);
7    }
8    else {
9      oref = g_atomic_int_add(&object→ref_count, -1);
10
11     if (oref == 1) {
12       object→klass→finalize(object);
13       g_type_free_instance(object);
14     }
15   }
16 }
17
18 void g_emblem_class_init(GEmblemClass *klass) {
19   klass→finalize = g_emblem_finalize;
20 }
21
22 void g_emblem_finalize(GObject *object) {
23   GEmblem *emblem = (GEmblem*)object;
24   g_object_unref(emblem→icon);
25 }
```

Figure 5.13: Managed types example from glib-2.0 library

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 5.13 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 initial-

ized 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 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 5.8.

5.6.3 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.

5.7 Related Work

The C/C++ leak detector of Heine and Lam (2003) 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 only handling C++ objects) and (2) incorporate shared ownership via reference counting and inferred contracts on function pointers used in library code. Rayside and Mendel (2007) 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. (2011) 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. (2003) address ownership at the type level with work on ownership types. Müller and Rudich (2007) 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 (2010) 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 5.6, we do not verify the correctness of reference count handling in the libraries we analyze. Emmi et al. (2009) 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.

5.8 Evaluation

In this section, we evaluate the effectiveness of the analyses described in this chapter. As in section 4.6, we present results based on the functions in each library callable by users. This is a slight superset of the public API of each library, which is only established by convention. We first discuss the allocator and finalizer analyses, as their results are most valuable to analysis consumers when considered together. We then evaluate the ownership transfer and reference counting analyses.

5.8.1 Allocator and Finalizer Analyses

The allocators and finalizers inferred by our analyses are reported in table 5.1. Two general expected trends are clearly visible in this table:

- There are more allocators than finalizers. This makes sense since there are often multiple allocators for a single type, but only one finalizer.
- Both analyses are incomplete. Libraries with finalizers but no allocators, such as libatk-1 and libfreenect, indicate the impact of incompleteness in the allocator analysis. Likewise, libraries with allocators but no finalizers, such as libz and libXrender, suggest the scale of the impact of the incompleteness of the finalizer analysis. It is worth noting that the number of allocators and finalizers in a library is a small percentage of the total number of functions in the same library.

In the remainder of this subsection, we discuss interesting findings in select libraries.

libbz2

Both of the user facing allocators are identified properly. The other two allocators in table 5.1 are not part of the public API. The analysis is unable to automatically identify the finalizer for this library, `BZ2_bzclose`. This function delegates its work to two other finalizers, which are not user facing. Both of these helpers return early if they detect errors. This particular case could be automatically resolved with:

- a generalization of the error code identification analysis in chapter 3 to track error codes returned through output parameters, and
- a modification to make the finalizer analysis aware of, and robust to, error paths.

Table 5.1: Inferred Allocator and Finalizer Annotations

Library	**# Funcs**	**Allocators**	**Finalizers**
libacl	61	3	0
libarchive	267	8	5
libatk-1	254	0	1
libattr	26	0	0
libbz2	33	4	0
libcairo	379	2	2
libcrypto	3552	0	39
libcurl	334	8	7
libdbus-1	804	30	9
libexif	142	15	5
libexpat	80	2	3
libffi	23	0	0
libfontconfig	196	50	8
libfreenect	61	0	1
libfreetype	289	5	6
libfuse	188	13	9
libgdk-3	758	14	7
libgdk_pixbuf-2	121	8	1
libgio-2	1772	55	11
libglib-2	1529	224	39
libglpk	1072	66	2
libgobject-2	394	15	3
libgsl	3910	258	111
libgtk-3	4784	125	21
libical	1045	115	17
libjansson	96	5	2
libpango-1	406	58	12
libpcre	31	2	3
libpixman-1	128	5	0
libpng15	391	1	0
libsoup-2	530	41	6
libsqlite3	185	0	0
libssl	475	0	0
libusb-1	53	2	4
libX11	1179	47	23
libXau	8	3	1
libXdmcp	42	0	0
libxml2	1652	76	61
libXrender	45	2	0
libz	71	3	0
Total	27366	1265	419

libglpk

The GNU Linear Programming Toolkit (GLPK) (Makhorin, 2008) defines its own low-level memory allocator in order to support more precise memory allocation accounting than is possible with the standard `malloc` and `free`. This allocator allocates extra memory for each allocation request. The extra memory is used as an object header to record metadata and links the allocated chunks into a list that can be traversed by query functions. The allocator then returns a pointer into the middle of each memory allocation, which is all that the caller sees. This foils our allocator analysis in two ways:

- the linked list construction allows each allocation to escape, which our definition of allocators does not allow, and
- returning a pointer into the middle of an allocated chunk of memory does not fit our definitions.

Likewise, the associated custom finalizer function does not follow our definition of a finalizer. The entry for GLPK in table 5.1 reflects the analysis results after we provided a single manual annotation to the analysis describing this custom allocator/finalizer pair. Note that these low-level custom memory management functions are not exposed to library callers at all. The allocator and finalizer analyses identify approximately 40 allocators and 30 finalizers in all, but only a few of these are actually exposed to the user. Nevertheless, one manual annotation is sufficient to discover the few relevant user-facing memory management functions.

5.8.2 Transfer Analysis

This section evaluates the effectiveness of the ownership transfer analysis from section 5.4 on a suite of six open source libraries. The reference counting analysis is evaluated on a separate set of libraries because, for the most part, libraries using a reference counting discipline do not require the results of the ownership transfer

Table 5.2: 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

analysis since ownership in those libraries is made explicit in the reference counts. We focus on the reduction in manual annotation burden compared to relying solely on the results of an escape analysis. Table 5.2 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.

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. *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 5.2 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 contracts imposed by these default values. 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 5.2 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 5.5, the escape annotations are still useful as object lifetime documentation to the user. The pointer parameters that escape without any ownership transfer require the caller to understand the object lifetime discipline of the library. Constructs like the pinning operation described in section 5.4.2 could be used to manage lifetimes safely. The pinning helper would only need to be provided once with a library

binding generator and would work for all libraries.

Note that the transfer analysis requires an accurate view of the finalizer functions in a given library. 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.

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. 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.

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.

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 a more nuanced approach. 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

Table 5.3: Number of inferred reference counting annotations.

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

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.

```
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 5.14: Finalizer from exif library

5.8.3 Reference Counting Analysis

Table 5.3 summarizes the results of our reference counting analysis on a suite of nine open source libraries. The "Ref/Unref" column reports the number of inferred *IncRef* and *DecRef* function pairs, rather than single functions. 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 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 5.14 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 5.6.2 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.

6 CONCLUSIONS

We have presented techniques to reduce the effort required to generate idiomatic library bindings for high-level programming languages, thus making polyglot programming more accessible. First, we described an analysis to infer the set of error codes used in a library, along with the functions that return them. The inference bootstraps itself from knowledge of the errors returned by the dependencies of each library, with the errors reported by the C standard library as the basis. The analysis is unsound and incomplete and includes a statistical component that could be extended with more sophisticated machine learning approaches. As a result, it works best on larger libraries with more system interactions from which to generalize. These are precisely the libraries where automated annotation assistance is most valuable, because the manual annotation burden on these larger libraries is otherwise prohibitive. The annotations inferred by this analysis can be used to automatically generate library bindings that convert returned error codes into high-level language exceptions. These inferred annotations can also be useful as inputs to other analyses and verification tools, as well as simply providing extra explicit documentation to library users.

Second, we presented a set of analyses to elucidate the semantics of pointer parameters. Pointers are used to encode many high-level concepts that have no direct representation in C. None of these extended semantics are apparent from C type signatures. Determining more precise semantics for them allows binding generators to create safer and more convenient, or idiomatic, library bindings. Bindings can be made safer by inserting run time checks that trap errors in high-level languages before they cause crashes in unsafe code. In some languages, these checks can even be encoded in type systems and eliminate run time overhead. The results of these analyses can make library bindings more convenient to use by automatically managing data conversions between languages.

Finally, we described a suite of analyses to infer how libraries manage memory.

These analyses construct a notion of allocator functions and finalizer functions based on the standard C `malloc` and `free` functions. The allocator functions are those returning new initialized objects. Finalizer functions safely dispose of these objects. The other memory oriented analyses infer how ownership of objects allocated by C functions is transferred between the C library and its callers. Our analyses support both (1) the common case where objects have only a single owner and (2) the case where objects have multiple owners, with lifetimes mediated by reference counts. We show that this information is sufficient to generate idiomatic library bindings that allow high-level language garbage collectors to automatically manage resources allocated by C libraries.

While these techniques cannot completely automate the library generation process due to unsoundness and incompleteness, we have shown that they are effective and useful. The annotations inferred by the unsound and incomplete analyses are intended to be checked and corrected by a developer familiar with the library being analyzed. This represents some manual effort required to generate library bindings. However, the amount of work required is significantly less than other approaches to binding generation. The amount of work can also be amortized significantly across multiple library versions; as libraries evolve, most annotations can remain constant or apply to newly-added functions.

6.1 Guidelines for Library Writers

Our experience with this work suggests three design guidelines to make C libraries more amenable to polyglot programming and to our analysis techniques. These guidelines also apply to other low-level languages that expose a C-compatible calling convention (C++, Fortran, and others).

6.1.1 Error codes and values should be disjoint

To the greatest extent possible, error codes should be treated as different from values. Taken to an extreme, this could imply that error codes are their own

distinct type. Alternatively, simply choosing error code constants that are out of range of any constant that would be returned by any function in the library would also meet this guideline. This simpler approach would require less code restructuring.

Either transformation would make inferring error codes and the functions returning them simpler and more reliable. Error code values overlapping valid domain values contribute to incorrectly inferred error codes. In many libraries, the success code is defined to be `0`, which commonly appears legitimately in many other contexts. These changes would let us generalize the analysis and no longer require us to ignore boolean-typed functions. False returns would no longer overlap with success codes, and true returns would no longer pollute the set of inferred error codes. That said, boolean functions should generally not return error codes alongside their normal returns.

One approach to this guideline would be to use function return values only for error codes and success codes. Any other values could be returned through output parameters. There is some precedent for this approach: COM-based libraries, common in Windows, always return a success or error code in their return value (of type `HRESULT`). Many POSIX functions similarly only return a single distinct value, `-1`, to report errors. Additional information about the error can be found by inspecting `errno`. In the vast majority of POSIX functions, `-1` is never a valid value.

6.1.2 Non-trivial types should be abstract

As many types as possible should be treated abstractly. Types whose representations are exposed, or *concrete types*, should not require initialization or finalization. Keeping types abstract allows libraries to maintain and rely on stronger internal invariants for those types. This makes reasoning about library behavior easier. A side effect is that libraries exposing only abstract types require less defensive programming than those that expose internal representations to clients.

```
1  void exif_entry_free (ExifEntry *e)
2  {
3    if (!e) return;
4
5    if (e→priv) {
6      ExifMem *mem = e→priv→mem;
7      if (e→data)
8        exif_mem_free (mem, e→data);
9      exif_mem_free (mem, e→priv);
10     exif_mem_free (mem, e);
11     exif_mem_unref (mem);
12   }
13 }
```

Figure 6.1: Defensive cleanup code

Besides helping developers reason about their own libraries, minimizing the number of concrete types would help our analysis. For example, defensive programming frequently interferes with our finalizer analysis. The example shown in figure 6.1 checks to ensure that a pointer to private data is valid on line 5 before proceeding to finalize the object. Due to this extra check, the function does not meet our definition of a finalizer. Furthermore, `exif_entry_free` leaks memory if the check does fail because the call to finalize the object on line 10 is guarded by the defensive check of the private field. Moreover, the caller has no way to determine if the call to the finalizer succeeded or failed. If the type `ExifEntry` was kept completely abstract, the invariant `e→priv != NULL` could be guaranteed and this additional check would be unnecessary.

Fully abstract types carry two disadvantages: they cannot be stack allocated and cannot be embedded in other objects by value (only by pointer). This is an efficiency concern. However, concrete types with no finalizers do not pose any problems for our analysis and do not expose callers to memory leaks of the form demonstrated in the above example. Finalizers are required for types that could leak resources. It is not uncommon for concrete types that are intended

to be allocated on the stack to require initialization via some function call. This is distinct from an allocator because the initialization function does not return a new object. As long as the initialization function does not store objects requiring finalization in the target, no finalizer would be required. Thus, a large class of concrete types are compatible with this guidance.

Minimizing the number of concrete types has additional benefits. If all types are abstract, changes to `struct` layouts are not visible to library clients, making the application binary interface (ABI) more stable and resilient to change. Additionally, abstract types are easier to pass across language boundaries than concrete types. Concrete types can be passed by both value and pointer, while abstract types can only be referred to by pointer. Passing `struct` types by value through an FFI is more difficult, and not supported in every FFI implementation. This difficulty again lies in platform ABIs: the proper method for passing `struct` type varies by architecture, operating system, and the size of the `struct`. Some FFIs elide support for passing `struct`s by value for simplicity.

6.1.3 Resources should be reference counted

Our ownership model inference is effective for many libraries. However, it adds additional complexity and requires more thorough oversight from knowledgeable developers. This additional complexity and developer burden could be avoided if user-facing types are reference counted. Reference counts make object ownership shared and lifetimes obvious. The implicit ownership tracking enabled by our ownership transfer analysis is most easily applied to polyglot programs with only two languages. Implicit ownership in polyglot programs with more than two languages is much more difficult to coordinate. Reference counts make polyglot programs with more than three languages tractable.

6.2 Closing Thoughts

These guidelines share a common theme: abstraction (Liskov, 2010). The first guideline argues that error codes should be an abstract type in the best case. They should never be mixed with values of non-error code types. The weaker model, in which error code constants are disjoint from other constant values that appear in the library, is also useful. The second guideline very clearly calls for abstraction of data types. The third guideline calls for abstracting object ownership models through reference counts. These abstractions remove many complex details of the C calling convention from user-facing functions, aiding API design by providing principled reasons favoring certain design decisions. They also make library analysis, and thus polyglot programming, easier.

Polyglot programs exist today. They will become more common as new languages mature and existing codebases continue to grow. Automatically generating idiomatic library bindings significantly reduces the barrier of entry for polyglot programming and promotes code re-use. Furthermore, automatically generating the tedious and error-prone glue code required for polyglot programs can make polyglot programs safer than their C equivalents. The generated wrappers can consistently check for errors and prevent them from becoming exploitable undefined behavior. Re-using C code from high-level programming languages then allows developers to be more productive. Our work makes significant contributions to the understanding of this under-served programming discipline. Our analyses infer enough information to make safe and idiomatic library bindings, given review by developers familiar with the libraries under analysis. While we have not completely automated the process of generating library bindings, we have significantly improved it over the prior state of the art. Furthermore, we find that library authors can make their code more reusable, easier to reason about, and more accessible to high-level programming languages by maintaining strong abstraction boundaries. Minimizing the number of C language minutiae that are exposed to library clients can simplify both library analysis and use in polyglot programs.

A DISTRIBUTIVITY OF OUTPUT PARAMETERS

The transfer function for writes through pointers in the output parameter analysis is GEN/KILL and trivially distributive. Below, we analyze the transfer function for reads through pointers by cases.

$$
\begin{aligned}
f(\text{Unused}) \sqcap f(\text{Unused}) &\stackrel{?}{=} f(\text{Unused} \sqcap \text{Unused}) \\
\text{IN} \sqcap \text{IN} &\stackrel{?}{=} f(\text{Unused}) \\
\text{IN} &= \text{IN}
\end{aligned}
$$

$$
\begin{aligned}
f(\text{Unused}) \sqcap f(\text{IN}) &\stackrel{?}{=} f(\text{Unused} \sqcap \text{IN}) \\
\text{IN} \sqcap \text{IN} &\stackrel{?}{=} f(\text{IN}) \\
\text{IN} &= \text{IN}
\end{aligned}
$$

$$
\begin{aligned}
f(\text{Unused}) \sqcap f(\text{OUT}) &\stackrel{?}{=} f(\text{Unused} \sqcap \text{OUT}) \\
\text{IN} \sqcap \text{IN/OUT} &\stackrel{?}{=} f(\text{IN/OUT}) \\
\text{IN/OUT} &= \text{IN/OUT}
\end{aligned}
$$

$$
\begin{aligned}
f(\text{Unused}) \sqcap f(\text{IN/OUT}) &\stackrel{?}{=} f(\text{Unused} \sqcap \text{IN/OUT}) \\
\text{IN/OUT} \sqcap \text{IN/OUT} &\stackrel{?}{=} f(\text{IN/OUT}) \\
\text{IN/OUT} &= \text{IN/OUT}
\end{aligned}
$$

$$
\begin{aligned}
f(\text{IN}) \sqcap f(\text{IN}) &\stackrel{?}{=} f(\text{IN} \sqcap \text{IN}) \\
\text{IN} \sqcap \text{IN} &\stackrel{?}{=} f(\text{IN}) \\
\text{IN} &= \text{IN}
\end{aligned}
$$

$$
\begin{aligned}
f(\text{IN}) \sqcap f(\text{OUT}) &\stackrel{?}{=} f(\text{IN} \sqcap \text{OUT}) \\
\text{IN} \sqcap \text{IN/OUT} &\stackrel{?}{=} f(\text{IN/OUT}) \\
\text{IN/OUT} &= \text{IN/OUT}
\end{aligned}
$$

$$
\begin{aligned}
f(\text{IN}) \sqcap f(\text{IN/OUT}) &\stackrel{?}{=} f(\text{IN} \sqcap \text{IN/OUT}) \\
\text{IN} \sqcap \text{IN/OUT} &\stackrel{?}{=} f(\text{IN/OUT}) \\
\text{IN/OUT} &= \text{IN/OUT}
\end{aligned}
$$

$$
\begin{aligned}
f(\text{OUT}) \sqcap f(\text{OUT}) &\stackrel{?}{=} f(\text{OUT} \sqcap \text{OUT}) \\
\text{IN/OUT} \sqcap \text{IN/OUT} &\stackrel{?}{=} f(\text{OUT}) \\
\text{IN/OUT} &= \text{IN/OUT}
\end{aligned}
$$

$$
\begin{aligned}
f(\text{OUT}) \sqcap f(\text{IN/OUT}) &\stackrel{?}{=} f(\text{OUT} \sqcap \text{IN/OUT}) \\
\text{IN/OUT} \sqcap \text{IN/OUT} &\stackrel{?}{=} f(\text{IN/OUT}) \\
\text{IN/OUT} &= \text{IN/OUT} \\
f(\text{IN/OUT}) \sqcap f(\text{IN/OUT}) &\stackrel{?}{=} f(\text{IN/OUT} \sqcap \text{IN/OUT}) \\
\text{IN/OUT} \sqcap \text{IN/OUT} &\stackrel{?}{=} f(\text{IN/OUT}) \\
\text{IN/OUT} &= \text{IN/OUT}
\end{aligned}
$$

This dissertation has investigated and evaluated several facets of the automated library binding generation problem, focusing on C libraries. The analyses discussed are sufficient for generating idiomatic library bindings for many different types of library. Nonetheless, there remain many avenues for future research that could improve the precision, or expand the scope, of our analyses. This appendix will outline a few such research directions.

B.1 Inference through Structure Fields

Many analyses in this document, and similar analyses, could benefit from inferring properties of C **`struct`** fields as well as pointer parameters. Inferences over properties of aggregate fields could provide more information for any analysis. A significant benefit of this extra information would allow analyses to infer properties of pointer parameters that are stored into fields of aggregates in one call, but are used only much later, perhaps with intervening high-level language code executing. For example, if a pointer parameter `p` is stored into a **`struct`** field `f` in a call to function `a`, but not otherwise used in `a`, the analysis could still conclude that `p` is not nullable if `f` is used without being checked against `NULL` in function `b`. The analysis may or may not be modified to take into consideration any checks against `NULL` in function `a`.

B.2 Targeted Runtime System Concerns

The characteristics of runtime systems of many high-level languages present many opportunities to exploit more information about foreign functions that they call. This section discusses a few analyses that could help automated library binding generation tools generate safer library bindings for certain high-level languages.

B.2.1 Re-entrant Functions

Some C functions, particularly in older libraries that were developed before threading became commonplace, are not re-entrant. This means that two calls to the same function occurring simultaneously can interfere with one another. Simultaneous calls can be the result of two threads calling the same function. They can also occur if the execution of a function is suspended due to an interrupt where the handler calls the same function again. Functions can fail to be re-entrant if they use global or `static` data. This is a problem for high-level languages that have strong support for parallelism and concurrency. Even languages without such support require some amount of care in the presence of interrupts, if they are supported.

An analysis that identifies functions that are not re-entrant could allow generators to create library bindings that prevent re-entrancy errors. For example, they could serialize access to re-entrant functions by automatically acquiring a lock before they are called. Alternatively, a simpler and faster mechanism could simply terminate the program if a function that is not re-entrant is about to be re-entered. This information could also aid developers in library understanding tasks and guide refactorings to eliminate functions that are not re-entrant.

B.2.2 Blocking Functions

For languages with single-threaded run-time systems, or run-time systems that use a single global interpreter lock, calls to blocking foreign functions are of special interest. Examples include:

Javascript Implementations are single threaded

Python The reference implementation uses a global interpreter lock

In the case of single-threaded run-time systems, calls to blocking functions must be specially constructed to integrate with the event loop of the run-time to avoid stalling the entire system. In implementations dependent on a global

interpreter lock, the lock must be released right before a blocking native call and acquired right after. This allows other threads in the run-time to continue while only one is blocked. With an analysis to automatically identify blocking functions, based on the calls they transitively call, a library binding generator could automatically create safe blocking calls.

B.3 API Usage Enforcement

Many libraries present an interface composed of functions designed to be called in some well-defined pattern. A common and general pattern involves library specific initialization, use, and then cleanup. Some libraries may have even more specific usage patterns that can be statically defined. If correct library usage patterns can be described using regular or context-free languages, it may be possible to arrange for generated library bindings to enforce correct usage dynamically.

For each library, generated bindings could maintain an automaton and modify its state just before issuing each call into the library. If the call would violate the behavior prescribed by the language of safe library calls, execution could be halted. A similar automaton could be maintained for each object allocated by a library, if the usage semantics can be defined on a per-object basis. Per-object automata would improve the enforcement granularity. This mechanism could be enabled for development to aid developers. It could then be disabled in release builds for efficiency. This mechanism would act like an inline reference monitor (Erlingsson and Schneider, 2000) with correctness as its goal, rather than security. It would be possible to impose some types of security constraints, as well as correctness constraints, on generated library bindings.

B.4 Interactivity and Inference Assistance

This work has focused on inferring as much as possible about libraries without human intervention beyond a few manual annotations provided as input. The results of many analyses could be improved with some limited library developer interaction. For example, being able to ask a knowledgeable library user if a particular constant is a success code would be valuable. More generally, having developer input to disambiguate unusual functions that perturb our analyses could make them more robust to unusual constructs in code. The case study of libusb in section 3.3.1 described an unusual function for which user feedback could have improved the analysis results. Most improvements could be accomplished by providing manual annotations. However, an interactive approach could more easily guide users to the most effective code locations on which to focus their efforts.

The described developer interaction could take many forms, with a workflow in the style of an interactive proof assistant. One important goal of work in this area would be to minimize the number of queries issued to developers. Work on generating minimal queries through abductive inference (Dillig et al., 2012) could be the starting point of these interactive extensions.

REFERENCES

Alpern, Bowen, Mark N. Wegman, and F. Kenneth Zadeck. 1988. Detecting equality of variables in programs. In *Popl*, ed. Jeanne Ferrante and P. Mager, 1–11. ACM Press.

Andersen, Lars Ole. 1994. Program Analysis and Specialization for the C Programming Language. Ph.D. thesis, University of Cophenhagen.

Ashley, J. Michael, and R. Kent Dybvig. 1994. An efficient implementation of multiple return values in scheme. In *Lisp and functional programming*, 140–149.

Beazley, David M. 2002. An extensible compiler for creating scriptable scientific software. In *International conference on computational science (2)*, ed. Peter M. A. Sloot, Chih Jeng Kenneth Tan, Jack Dongarra, and Alfons G. Hoekstra, vol. 2330 of *Lecture Notes in Computer Science*, 824–833. Springer.

Beazley, David M., and Peter S. Lomdahl. 1997. Building flexible large-scale scientific computing applications with scripting languages. In *Ppsc*. SIAM.

Boyapati, Chandrasekhar, Barbara Liskov, and Liuba Shrira. 2003. Ownership types for object encapsulation. In *Popl*, ed. Alex Aiken and Greg Morrisett, 213–223. ACM.

Chang, Chih-Chung, and Chih-Jen Lin. 2011. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology* 2: 27:1–27:27. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

Cheng, Ben-Chung, and Wen-Mei W. Hwu. 2000. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Pldi*, ed. Monica S. Lam, 57–69. ACM.

Choi, Jong-Deok, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. 2003. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Trans. Program. Lang. Syst.* 25(6):876–910.

Dillig, Isil, Thomas Dillig, and Alex Aiken. 2012. Automated error diagnosis using abductive inference. In *Pldi*, ed. Jan Vitek, Haibo Lin, and Frank Tip, 181–192. ACM.

Elwazeer, Khaled, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. 2013. Scalable variable and data type detection in a binary rewriter. In *Pldi*, ed. Hans-Juergen Boehm and Cormac Flanagan, 51–60. ACM.

Emmi, Michael, Ranjit Jhala, Eddie Kohler, and Rupak Majumdar. 2009. Verifying reference counting implementations. In *Tacas*, ed. Stefan Kowalewski and Anna Philippou, vol. 5505 of *Lecture Notes in Computer Science*, 352–367. Springer.

Engler, Dawson R., David Yu Chen, and Andy Chou. 2001. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Sosp*, 57–72.

Erlingsson, Úlfar, and Fred B. Schneider. 2000. Irm enforcement of java stack inspection. In *Ieee symposium on security and privacy*, 246–255. IEEE Computer Society.

Foster, Jeffrey S., Manuel Fähndrich, and Alexander Aiken. 1999. A Theory of Type Qualifiers. In *Pldi*, 192–203.

Furr, Michael, and Jeffrey S. Foster. 2005. Checking Type Safety of Foreign Function Calls. In Sarkar and Hall (2005), 62–72.

Gabriel, Richard P., David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., eds. 2007. *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*. ACM.

GNU Project. 2013. The GNU C Library. http://www.gnu.org/software/libc/.

gobject. 2011. GObject Introspection. http://live.gnome.org/GObjectIntrospection.

Godefroid, Patrice, Nils Klarlund, and Koushik Sen. 2005. Dart: directed automated random testing. In Sarkar and Hall (2005), 213–223.

Hackett, Brian, Manuvir Das, Daniel Wang, and Zhe Yang. 2006. Modular checking for buffer overflows in the large. In *Icse*, ed. Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, 232–241. ACM.

Heine, David L., and Monica S. Lam. 2003. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Pldi*, 168–181. ACM.

Heller, Thomas. 2008. ctypeslib – useful additions to the ctypes FFI library. http://pypi.python.org/pypi/ctypeslib/.

Horspool, R. Nigel, and Jan Vitek. 1992. Static analysis of postscript code. In *Iccl*, ed. James R. Cordy and Mario Barbacci, 14–23. IEEE.

ISO. 2006. C# programming language. ISO ISO/IEC 23270:2006(E), International Organization for Standardization, Geneva, Switzerland.

Jaroszewicz, Szymon. 2008. ctypesGSL. http://www.cs.umb.edu/~sj/ctypesGsl/.

Khedker, Uday P., Amitabha Sanyal, and Amey Karkare. 2007. Heap reference analysis using access graphs. *ACM Trans. Program. Lang. Syst.* 30(1).

Kildall, Gary A. 1973. A unified approach to global program optimization. In *Popl*, ed. Patrick C. Fischer and Jeffrey D. Ullman, 194–206. ACM Press.

Lattner, Chris, and Vikram S. Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Cgo*, 75–88. IEEE Computer Society.

Lee, Byeongcheol, Ben Wiedermann, Martin Hirzel, Robert Grimm, and Kathryn S. McKinley. 2010. Jinn: Synthesizing Dynamic Bug Detectors for Foreign Language Interfaces. In *Pldi*, ed. Benjamin G. Zorn and Alexander Aiken, 36–49. ACM.

Leshchinskiy, Roman. 2013. The Haskell vector Package. `http://hackage.haskell.org/package/vector`.

Li, Lian, Cristina Cifuentes, and Nathan Keynes. 2011. Boosting the performance of flow-sensitive points-to analysis using value flow. In *Sigsoft fse*, ed. Tibor Gyimóthy and Andreas Zeller, 343–353. ACM.

Liskov, Barbara. 2010. The power of abstraction - (invited lecture abstract). In *Disc*, ed. Nancy A. Lynch and Alexander A. Shvartsman, vol. 6343 of *Lecture Notes in Computer Science*, 3. Springer.

Makhorin, Andrew. 2008. GLPK (GNU linear programming kit). `http://www.gnu.org/software/glpk/`.

Matosevic, Ivan, and Tarek S. Abdelrahman. 2012. Efficient bottom-up heap analysis for symbolic path-based data access summaries. In *Cgo*, ed. Carol Eidt, Anne M. Holler, Uma Srinivasan, and Saman P. Amarasinghe, 252–263. ACM.

Müller, Peter, and Arsenii Rudich. 2007. Ownership transfer in universe types. In Gabriel et al. (2007), 461–478.

Negara, Stas, Rajesh K. Karmani, and Gul A. Agha. 2011. Inferring ownership transfer for efficient message passing. In *Ppopp*, ed. Calin Cascaval and Pen-Chung Yew, 81–90. ACM.

NumPy. 2013. NumPy. `http://www.numpy.org/`.

Pearce, David J., Paul H. J. Kelly, and Chris Hankin. 2007. Efficient field-sensitive pointer analysis of c. *ACM Trans. Program. Lang. Syst.* 30(1).

Ravitch, Tristan. 2012. Fix an error test on the result from fread. `https://github.com/libarchive/libarchive/pull/29`.

Rayside, Derek, and Lucy Mendel. 2007. Object ownership profiling: a technique for finding and fixing memory leaks. In *Ase*, ed. R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer, 194–203. ACM.

Reppy, John H., and Chunyan Song. 2006. Application-specific Foreign-interface Generation. In *Gpce*, ed. Stan Jarzabek, Douglas C. Schmidt, and Todd L. Veldhuizen, 49–58. ACM.

Rubio-González, Cindy, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. 2009. Error Propagation Analysis for File Systems. In *Pldi*, ed. Michael Hind and Amer Diwan, 270–280. ACM.

Sagiv, Shmuel, Thomas W. Reps, and Reinhard Wilhelm. 2002. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.* 24(3): 217–298.

Salcianu, Alexandru, and Martin C. Rinard. 2001. Pointer and escape analysis for multithreaded programs. In *Ppopp*, ed. Michael T. Heath and Andrew Lumsdaine, 12–23. ACM.

Sarkar, Vivek, and Mary W. Hall, eds. 2005. *Proceedings of the acm sigplan 2005 conference on programming language design and implementation, chicago, il, usa, june 12-15, 2005*. ACM.

Sidiroglou, Stelios, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D. Keromytis. 2009. Assure: automatic software self-healing using rescue points. In *Asplos*, ed. Mary Lou Soffa and Mary Jane Irwin, 37–48. ACM.

Smolinski, Brent A., Scott R. Kohn, Noah Elliott, and Nathan Dykman. 1999. Language Interoperability for High-Performance Parallel Scientific Components. In *Iscope*, ed. Satoshi Matsuoka, R. R. Oldehoeft, and Marydell Tholburn, vol. 1732 of *Lecture Notes in Computer Science*, 61–71. Springer.

Steele, Guy L. 1990. *Common LISP: the Language*. Digital Press.

Tan, Gang, and Greg Morrisett. 2007. Ilea: Inter-language Analysis across Java and C. In Gabriel et al. (2007), 39–56.

Wegiel, Michal, and Chandra Krintz. 2010. Cross-language, Type-safe, and Transparent Object Sharing for co-Located Managed Runtimes. In *Oopsla*, ed. William R. Cook, Siobhán Clarke, and Martin C. Rinard, 223–240. ACM.

Weimer, Westley, and George C. Necula. 2004. Finding and preventing run-time error handling mistakes. In *Oopsla*, ed. John M. Vlissides and Douglas C. Schmidt, 419–431. ACM.

———. 2005. Mining temporal specifications for error detection. In *Tacas*, ed. Nicolas Halbwachs and Lenore D. Zuck, vol. 3440 of *Lecture Notes in Computer Science*, 461–476. Springer.

Whaley, John, and Martin C. Rinard. 1999. Compositional pointer and escape analysis for Java programs. In *Oopsla*, ed. Brent Hailpern, Linda M. Northrop, and A. Michael Berman, 187–206. ACM.