Array Length Inference for C Library Bindings

Alisa J. Maas
University of Wisconsin–Madison
Madison, WI, USA
ajmaas@cs.wisc.edu

Henrique Nazaré
Universidade Federal de Minas Gerais
Belo Horizonte, Brazil
hnsantos@dcc.ufmg.br

Ben Liblit
University of Wisconsin–Madison
Madison, WI, USA
liblit@cs.wisc.edu

ABSTRACT

Simultaneous use of multiple programming languages (*polyglot programming*) assists in creating efficient, coherent, modern programs in the face of legacy code. However, manually creating bindings to low-level languages like C is tedious and error-prone. We offer relief in the form of an automated suite of analyses, designed to enhance the quality of automatically produced bindings. These analyses recover high-level array length information that is missing from C's type system. We emit annotations in the style of GObject-Introspection, which produces bindings from annotations on function signatures. We annotate each array argument as terminated by a special sentinel value, fixed-length, or of length determined by another argument. These properties help produce more idiomatic, efficient bindings. We correctly annotate at least 70% of all arrays with these length types, and our results are comparable to those produced by human annotators, but take far less time to produce.

CCS Concepts

•**Software and its engineering** → **Automated static analysis; Software libraries and repositories;** *Data types and structures; Software maintenance tools;* •**Theory of computation** → *Type structures;* Pattern matching;

Keywords

FFI, foreign function interfaces, bindings, libraries, static analysis, type inference

1. INTRODUCTION

In modern programs, writing code in a single language may not always suffice. Developers may wish to write new code in one programming language yet use legacy code written in another, or may wish to switch among languages depending on the task at hand. This leads programmers to produce *polyglot programs* that mix multiple languages in a single application. *Foreign function interfaces* (FFIs) support polyglot developers by letting high-level languages call into low-level languages through a series of *library bindings*. These bindings can hide the tedious details of converting data types from one language to another. In the context of a cross-language function call, the *host* language is the language supporting the callee, and the *guest* language is the language supporting the caller.

A well-written binding does more than just hide low level details of polyglot programming. It additionally exposes low-level language functions in a way that is consistent with the style and idioms of the high-level-language. For example, a C function that accepts an array usually also requires the array's length as a separate argument. A well-written, idiomatic binding hides such details, freeing the programmer to simply pass the array.

However, creating bindings manually is time-consuming and tedious. Additionally, human-created bindings frequently contain errors (Section 5.3.2), resulting in a scarcity of high-quality bindings. We are concerned with creating high-quality bindings to C, a popular target for language bindings. However, the C type system lacks high-level type information, complicating the automatic production of high-quality bindings. For example, most high-level languages clearly distinguish pointers (references) from lists, and the representation of a list includes its length. By contrast, C conflates arrays with pointers. A C array is simply a raw pointer to allocated memory that may extend beyond a single element, and the length of an array is not stored as part of its run-time representation. Even a C string is represented merely as the **char**$*$ pointing to its first character.

Thus, C developers are left on their own to determine how large a given array is, and may adopt different strategies in different functions. Three idiomatic strategies are particularly common:

1. The array ends with a special *sentinel value* that can never appear as a regular array element. Such arrays are considered to be *sentinel-terminated*. Correct C programs should not read past this sentinel value. C strings are the most common example. Each string in C is represented as an array of **char** ending with the sentinel character '\0', or ASCII NUL.

2. The length is stored as some other value maintained alongside the array itself. For example, a function may take two arguments: one for the array, and one for the length. Likewise, a structure might store an array and its length in a pair of fields.

3. The length of the array is a constant. A fixed-length array of size k requires an implicit agreement between the caller and the callee: the callee provides an array of at least size k, and the caller never accesses more than k elements from the array.

In C, there is no way for a function to verify that it has been given an array argument of the correct length. A library binding written in a high-level language could perform this task. Ideally, a library binding for a function accepting a C pointer should allow the caller to present a high-level array or string when appropriate. Our goal is to automate the production of such language bindings.

ASE'16, September 3–7, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-3845-5/16/09...$15.00
http://dx.doi.org/10.1145/2970276.2970310

```
def foo(array, string, fixedLen):
    x = c_expectLenArg(array, len(array))
    nulSafeArray = string.replace('\0', '')
    nulSafeArray.append('\0')
    y = c_expectNulTerm(nulSafeArray)
    z = c_expectFixedLen(fixedLen)
    return (x, y, z)
```

Listing 1: Calls using language bindings from Python to C without length inference

```
def foo(array, string, fixedLen):
    x = c_expectLenArg(array)
    y = c_expectNulTerm(string)
    z = c_expectFixedLen(fixedLen)
    return (x, y, z)
```

Listing 2: Calls using language bindings from Python to C with length inference

The remainder of this paper is organized as follows. Section 2 establishes the motivations for automating annotation inference, and describes the annotation system that consumes our analysis results. Section 3 reviews related work to set the context for our novel approach. In Section 4 we formalize each length idiom, and present our approach in detail (for each distinct length idiom as well as for combining results across uses). Experimental evaluations in Section 5 assess the effectiveness of our implementation when applied to multiple real-world libraries. Section 6 discusses options for future work and Section 7 concludes.

2. MOTIVATION

Our work automatically recovers high-level information about array arguments in C library functions, enabling automatic production of high-quality, idiomatic language bindings to C. Specifically, we provide analyses that recover C array argument lengths from LLVM [12] bitcode. Prior work approaches the problem of determining the lengths of C arrays with the motivation of discovering memory vulnerabilities in libraries, such as buffer overflow violations. Our focus on language binding generation allows us to focus on extracting programmer *intent* rather than discovering buggy code. This enables us to recover more information about the intended length idiom, which improves language bindings by freeing the caller to pass a high-level string or array. Length information required by the callee can be extracted by the binding, not the developer. This makes the binding more intuitive, and lessens the risk that the user of the library binding might accidentally provide incorrect length information.

Listings 1 and 2 show three calls to language bindings from Python to C. Assume that array is a list with arbitrary length, string is a NUL-terminated string, and fixedLen is an array of exactly length 4. In each example, the binding hides some of the more frustrating parts of making an external call: allocating space for the array and copying all of the elements over. Notice that Listing 1 contains an additional loop and allocation in the form of the string.replace call. The string must be NUL-terminated and not contain embedded NUL characters, as the low-level C function expects. Without high-level array type information available in the binding, the user is forced to handle this manually. If the user is unsure whether the string ends with a NUL or contains embedded NUL characters, she might have to manually copy the characters to a separate array to ensure this.

In Listing 2, the user directly passes the arrays and string without extracting length information. In Listing 1, she must manually pass the length of each array, even though the Python object representing each array maintains length information. Extracting the length information on the Python side is straightforward; the difficulty lies in determining what length information the C code requires.

Although Listings 1 and 2 call functions whose names make the expected length information abundantly clear, this is often not obvious from the API. Programmers who wish to call a C function must first search for documentation, which may or may not give an indication as to expected length conventions. Worse, in many libraries, this documentation is sparse, out-of-date, or non-existent. Developers may be forced to examine the source code by hand, searching for evidence of one length convention or another. We address this hidden work involved in manually creating a cross-language call such as the one in Listing 2. To that end, we automate this process using static analysis. Recovering high-level type information about C arrays lets us produce language bindings that are more intuitive for users of high-level languages.

In the case of fixed-length arrays, we offer some benefit beyond a more intuitive binding: we can produce a more efficient binding by stack-allocating arrays wherever possible. GObject-Introspection supports annotations providing memory ownership information, and these annotations can be inferred using work by Ravitch and Liblit [19] (see Section 3). This gives GObject-Introspection the ability to know when it is safe memory management to stack-allocate arrays. However, only fixed-length arrays may be stack-allocated in C. Stack allocated arrays avoid memory leaks due to incorrect library usage and also reduce heap churn, simplifying the job of the garbage collector. Bindings with stack-allocated arrays are also more amenable to further analysis than bindings that use the heap.

One difficulty in automatically extracting length information concerns the availability of code which uses the library, or *client code*. Client code is a natural way to discover information about the lengths of arrays, both at allocation points and at library API call sites. Unfortunately, many libraries that would benefit from an automated language binding generator do not have easily accessible client code. It might be possible to use test code instead. However, among the six libraries in our evaluation, gck has no test suite, and libssh2's minimal tests do not cover its entire API. The remaining libraries have tests, but we cannot speak to their thoroughness. Further, we expect that production of language bindings is most helpful early in a library's development.At this time, tests are likely to be incomplete or even missing. Thus, we do not assume client code will be present.

Our high-level goal is to create more automatic, intuitive bindings: more like those in Listing 2 than those in Listing 1. We expect this to reduce frustration and ease the learning curve associated with polyglot programming, for the developer producing the binding and the developer using the binding. If creating intuitive bindings is made easier for developers, more developers will create language bindings. If more intuitive language bindings exist, more programmers will be able to effectively make use of polyglot programming.

To that end, we emit annotations in a format read by GObject-Introspection [9]. GObject-Introspection provides a suite of tools that read in a series of annotations to provide an automatic, idiomatic language binding to C. GObject-Introspection can produce language bindings to C from many high-level languages, including Python, Java, Perl, and others. In addition to producing language bindings to each of these languages, GObject-Introspection streamlines the process of producing language bindings, making it possible to create language bindings from an arbitrary language more efficiently. Examples in this paper assume that the guest language is Python. However, GObject-Introspection's annotations are more general, and

our tool inherits its generality. While we are limited to GObject-Introspection's annotations, their annotations are fairly extensive, and already have many users. The utility of a ready-made user base and binding generator far outweighs the modest improvement in precision we could get by creating our own system of more complex annotations. Producing length annotations automatically bypasses some of the tedious work involved in writing language bindings, which saves developer time. Further analyses could be combined with ours to create the full set of GObject-Introspection annotations, which are not limited to just array length annotations.

3. RELATED WORK

Ravitch et al. [20] automatically generate bindings based on static analysis of C, while Ravitch and Liblit [19] analyze memory ownership in C libraries to produce bindings that correctly handle memory management. Our work extracts array length information, not memory management models, from C. It could be used in cooperation with these to produce better bindings. SALInfer [11] statically analyzes C, in part to detect potential buffer overflows. SALInfer also produces annotations, including a "zterm" annotation for strings, which it detects by recognizing writes of NUL into buffers. SALInfer operates over a complete program, and therefore is guaranteed to have access to the source code that writes NUL into each sentinel-terminated array. We analyze library code, and therefore cannot assume that sentinel writes are visible to us. Furr and Foster [4] describe a pair of tools that ensure type-safety of OCaml-to-C and Java-to-C (JNI) bindings. These tools are complementary to ours, as they statically check produced FFIs for safety, whereas we automate part of the process of creating those FFIs. Lu et al. [15] perform access correlation in order to hunt concurrency bugs. In particular, they track constraint specification, which includes symbolic lengths of arrays. However, they focus only on globals and structure fields, in order to narrow in on concurrency bugs, while we are interested ultimately in arguments of arrays.

CCured [18] retrofits run-time bounds checks into C code to ensure memory safety. CCured identifies potentially unsafe accesses by using type inference rules that follow from physical subtyping and limited manual annotations. We share CCured's desire to use static inference to extend the limited type system of C. However, our ultimate goals differ, leading CCured to add run-time checks for potentially unsafe memory accesses. In contrast, we might ignore these accesses to extract high-level programmer intent. Further, CCured requires some hand-crafted annotations; we require none.

A host of other work attempts to recover length information in C, typically with the goal of statically detecting memory safety violations. Wies et al.'s shape analysis [24] relies on complex symbolic predicates to facilitate a precise approach. They require precision in order to accept only safe memory accesses. Dhurjati et al. [3]'s static analysis enforces memory safety without (programmer-created) annotations, run-time checks, or garbage collection. They provide a region analysis to accomplish this. Our approach is more heuristic, which may allow us to derive more information.

All of these approaches analyze complete programs, not library functions. They assume that the code being analyzed is untrustworthy, while we assume that library code is correct (at least in intent). A bug-hunting approach seeks inconsistencies in the way C length information is treated, and so can only determine that an array is used safely. Our high-level understanding of the length of an array does not require bug-free implementations. We extract developer *intent*, which may still be recognizable despite implementation errors.

Le and Soffa [13] detect user-specified faults, and use path-sensitive data to reduce the number of false positives presented to the user. They categorize potentially vulnerable statements into five types, allowing their users to focus on relevant statements. They recognize the burden on the programmer to provide length information and wish to automate this process. SoftBound [16] analyzes metadata created at run time in order to catch unsafe memory accesses. SoftBound uses static analysis to determine where to use metadata at run time, but they focus on program transformation. They attempt to find every potentially unsafe memory access. Further, they do not produce symbolic length types evident in the source code; length information is stored at run time in metadata. We seek a purely static approach, as we do not assume a complete program.

Rugina and Rinard [21] use an interprocedural bounds analysis to determine memory safety, and has a wide variety of applications. Their approach is similar to that of the symbolic range analysis performed by Nazaré et al. [17], which we use to compute upper and lower bounds of array indices in our tool. Nazaré et al.'s approach is very lightweight, and appears to scale well to large programs, while Rugina and Rinard's results indicate that there may be issues with scaling to larger programs, such as the libraries we intend to analyze.

Alves et al. [1] provide an optimization technique to disambiguate pointers at run time. Their focus is towards producing superior optimized code, and to this end they transform the code to make use of run-time information. Although disambiguation of pointers is very useful for our algorithm, we take a static approach, and thus cannot make use of dynamic information.

SWIG [2] is a very popular tool providing a different set of bindings from GObject-Introspection, and theoretically could benefit from our provided annotations as well. However, at this time, SWIG does not support annotations identifying pointers as arrays, nor handle the lengths of arrays. Thus we target GObject-Introspection instead.

4. APPROACH

This section is laid out as follows. Section 4.1 describes the formal definitions of each of our length properties, and Section 4.2 introduces the assumptions we leverage to approximate these length properties. Sections 4.3 to 4.5 lay out the analyses used to recover length information, and Section 4.6 describes our method for merging length properties when more than one strategy appears to be used for encoding the length. Section 4.7 discusses expanding our analysis to structure fields. Finally, Section 4.8 explains sources of false positives and true negatives in our analyses.

4.1 Formal Definitions

Let a be one dynamic instance of a zero-based array in one execution of a function in one particular run of the code under analysis. Let $access(a, i)$ be true if this specific run ever accesses a at element i. Let $allocated(a)$ be the total number of elements allocated in the block of memory containing a. Note that $allocated(a) \geq 0$ in all cases. Then define $memsafe(a)$ as $\nexists i \geq allocated(a)$ such that $access(a, i)$. This is the basic **memory safety property**, which requires that a never be accessed beyond its allocated bounds.

We now define the length of an array argument a in the context of a particular execution of function f. In each of the following cases, assume first that $memsafe(a)$. Then:

Case 1: Let k be the minimum non-negative integer such that $access(a, i) \rightarrow i < k$. Then a **has fixed-length** k**.**

Case 2: Let n be an argument to f. If $access(a, i) \rightarrow i < n$, then a **has symbolic-length** n**.**

Case 3: Let ω be some sentinel value. If $\exists n \mid a[n] = \omega \wedge \forall 0 \leq i < n, a[i] \neq \omega \wedge \forall i > n, \neg access(a, i)$, then a **is sentinel-terminated by** ω**.**

Jointly, we refer to these as the *formal length properties* of an array. For a to have a property statically, it must have that property in every possible execution of f. Thus, the valid static properties of a are the intersection of the properties across all executions of f under all possible inputs. It is possible for an array to have more than one of these length types simultaneously. An array might have a fixed-length, and always end with a NUL. Likewise, perhaps an array has a symbolic-length whose actual value is always a constant. In general, an array's length may be some function of other symbolic or constant values. However, empirically, this is very uncommon, and we do not address this generalization.

All three of the above cases are undecidable in general. For example, $access(a, i)$ is quantified over all possible runs, on all possible inputs, while $allocated(a)$ requires knowing the exact sizes of arrays, including those dynamically allocated. In the context of library APIs lacking client code, this becomes even more challenging: the allocation points may not even be present. These definitions serve as useful Platonic ideals: perfect but unattainable. With these in mind, we design static approximations that sacrifice soundness and completeness in exchange for decidability and greater utility.

4.2 Key Ideas

We address the problem of extracting high-level properties about the static lengths of pointers representing arrays in C. In particular, we recover whether each array argument to a C function is terminated by a sentinel value, or discover the symbolic or constant value representing the length. Our approach for doing so necessarily approximates the (undecidable) formal length properties. Our approach at times over-approximates and at times under-approximates these properties. In order to more completely analyze libraries, we make **three key assumptions**, which introduce these approximations.

1. We assume that functions will not be intentionally obfuscated, and that **the developer intends for each array argument to have at most one formal length property**.

2. We assume that library code treats an argument like a sentinel-terminated, symbolic-length or fixed-length array only if that matches programmer intent. Due to this, **if an argument is ever treated as if it has a length property, it has that property**. The formal definitions (Section 4.1) only ascribe a property to an array if it must have that property across all possible executions. We ascribe properties to arrays if they have that property in some execution. Our reasons for doing this are twofold: computing the exact array length properties is undecidable in the general case; and due to assumption 1.

3. We assume the memory safety property discussed in Section 4.1. That is, we assume that **all accesses to elements from an array are safe**. Leveraging this assumption, we recover programmer intent of array length, even in code without the memory safety property (where behavior is technically undefined). **While bug-hunting approaches analyze the code you have, we analyze the code you think you have.**

We do *not* assume that we have any access to client code that uses a given library. Analyzing client code would allow us to make use of arguments to malloc indicating the actual size of the array. However, we intend for our tool to be useful to developers early in the development process, when there may not yet be any client code to analyze. Further, the amount of work required for the developer to find client code, verify that it uses the library code as intended, install the client code and then run our tool may be prohibitive, and our work seeks to make the process as easy as possible for the library developer. Thus, we do not currently analyze client code.

```
int symbolicLength(int *array, int y)
{
    if (array[3] < 0)
        return array[0];

    int sum = 0;
    for (int i = 0; i < y; i++)
        sum += array[i];

    return sum;
}
```

Listing 3: Example function with an argument of symbolic length

4.3 Symbolic Range Analysis

Our techniques (especially those in Section 4.4) rely heavily on determining upper bounds on indices into arrays. We accomplish this using static range analysis. Range analyses attempt to statically infer intervals that conservatively encompass all values a given program variable can assume. The range analysis we use, which is described and implemented by Nazaré et al. [17], handles only integer values and allows interval expressions to be symbolic.

For a practical explanation, consider Listing 3, with three integer variables (y, sum, and i) and one array variable (array). Let $[\![y]\!]$ represent the statically-inferred interval across which y ranges, and likewise for $[\![sum]\!]$ and $[\![i]\!]$. y is regarded as an input, as are all integer parameters, since nothing is known about the values they can assume. In a numeric range analysis, this would typically mean that it resides at the top of the interval lattice, which denotes a complete absence of information. This can be represented by $[-\infty, \infty]$. On a symbolic lattice, however, symbolically representing variable bounds is possible. So, while very little is known about the actual numeric values y assumes, denoting its interval as $[y, y], y \in \mathbb{N}$ is valid and is what the symbolic range analysis does. Naturally, since i is bounded below by 0 and bounded above by y, $[\![i]\!] = [0, y]$. Through variable renaming, the analysis also infers that inside the **for** loop, $[\![i]\!] = [0, y - 1]$. The variable sum is repeatedly summed with unknown values, so the analysis gives it an abstract state of $[-\infty, \infty]$.

Range analyses have a variety of applications, such as static branch prediction or proving safety with regard to memory and integer overflow. Nazaré et al. use their analysis to check array subscripts against the size of the arrays themselves, in an attempt to statically prove the safety of load and store operations; i.e., the memory safety property for loads and stores. We instead use subscript intervals to infer the intended lengths of the arrays they index, as we explain in Section 4.4.

4.4 Symbolic- and Fixed-Length Detection

At a high level, we infer that array argument array has symbolic length y if at most the first y elements are accessed from it. Note the parallel to our definition in Section 4.1. We are attempting to identify exactly the length in case 2. Consider the function in Listing 3. This function accesses the elements 0 through y – 1 from array. Note that array could actually have more than y elements; this code would still obey the memory safety property if y were smaller than the actual length of array. We exploit key assumption 2 when we say that array has symbolic length y. We cannot recover $allocated(array)$, but we can recognize the programmer's intended length.

To find symbolic lengths, we consider the possible range of values for each index into each pointer argument, array, in a function, f.

```
int symbolicLoop(int *array, int y)
{
    int *end = array + y;
    int sum = 0;
    while (array < end) {
        sum += *array;
        array++;
    }
    return sum;
}
```

Listing 4: Example function with an argument of symbolic length containing a specialized loop

```
guint g_str_hash (gconstpointer v)
{
    const signed char *p;
    guint32 h = 5381;

    for (p = v; *p != '\0'; p++)
        h = (h << 5) + h + *p;

    return h;
}
```

Listing 5: Real-world function with an argument sentinel-terminated by NUL, taken from glib

Assume that y is some integer argument to f. For our analysis to conclude that array has symbolic length y, some index must have upper bound y − 1, and all other indices must either have a smaller upper bound or a constant upper bound. This represents another departure from our definition of symbolic-length from Section 4.1. We apply our domain knowledge in order to assume that y will be larger than any constant in the general case. We have not empirically found any false positives arising from this assumption. In Listing 3, we can see that the **for** loop will access precisely elements 0 through y − 1, while element 3 is accessed outside the loop.

With a more sophisticated analysis, we might be able to recover information such as "array has at least length 3 and at least length y". However, the utility of this for producing language bindings is limited to dynamic checks that the binding is being used correctly. While it provides some benefit, it does not directly make the production of language bindings more automatic or more intuitive. Further, GObject-Introspection, our target binding generator, does not currently support annotations that are complex enough to express this. Notice that if our goal were to statically check for the memory safety property, such information would be of great use. This would allow us to infer that in order for the array access to be correct, array must have size at least 3. We could then prove that array always has at least size 3, or find some input where the size is less than 3.

We use Nazaré et al.'s Symbolic Range Analysis tool (*SRA*) [17] to determine upper bounds of array indices. However, SRA only computes the upper bounds of *integer* values; it does not handle pointers ranges. We first transform pointer arithmetic and pointer comparisons into equivalent array-offset indices, which SRA can then analyze. This transformation replaces any code that increments a pointer, array, with code incrementing an index, i, by the same amount. Any operations accessing element j from array are then transformed to access element j + i. (In most cases, j is 0.) This transformation is interesting in two ways. First, it is an example of a de-optimization done to make the code easier to reason about and analyze. Strength reduction, a common class of compiler optimization, may generate code that increments an array (interpreted as a pointer to its data) rather than using array-offset indices. We essentially reverse this optimization in order to make more effective use of SRA. Second, our transformation pass may produce code that is less efficient than the original. However, this is unimportant as the code is discarded after analysis without being run.

We take an even more lenient approach to determining whether an array may have a symbolic length in the presence of loops. For each loop, we find code that compares the address of some element of the array to a fixed offset from the array (y), and branches out of the loop upon reaching that offset. If each iteration of the loop must complete such a check, then array has symbolic length y, regardless of what happens outside the loop. For example, in Listing 4, each iteration of the **while** loop must compare the current value of array to the initial value of array + y. The **while** loop terminates once they are equal. This approach is a heuristic; it causes us to over-approximate the set of symbolic-length arrays compared to the definition of symbolic-length from Section 4.1. This arises because of key assumption 1. In practice, it appears that C programmers follow this assumption, even though the type system does not require this. We have found no false positives resulting from this heuristic in our empirical evaluation.

Fixed-length arrays are a special case of symbolic-length arrays, where every offset from the array is constrained to be a constant. If a non-constant offset from the array is ever accessed, then the array cannot be fixed-length. The similarity in our approaches here mirrors the similarity in our definitions of fixed-length and symbolic-length: Cases 1 and 2 in Section 4.1.

4.5 Sentinel-Terminated Detection

Per Section 4.1, an array is terminated by a sentinel value if that value lies at the *logical* end of the array. Note that this does not necessarily mean that the sentinel value lies at the end of allocated memory. Rather, any reads past the sentinel value have no semantic interpretation. Since we assume correct code, we expect not to see any such reads. Listing 5 shows a real-world string hash function that accepts a logical string, and treats it accordingly in the loop. To identify sentinel-terminated arrays, we search for arrays that are never read past the sentinel character. In Listing 5, the sentinel character is '\0', or ASCII NUL, and after the function processes a NUL character, it never reads another element from the array.

Our analysis for sentinel-terminated arrays leverages loop structures in order to detect the sentinel-terminated property. Consider a function, f, with pointer argument array. We examine each natural loop that accesses offsets of array (directly or transitively), and compute its set of mandatory sentinel checks of array.

Let the entry of the loop be L_{entry}. Let $check(\text{array}, i, \omega, b)$ be an access of array at offset i, comparing the value at this offset to ω with Boolean result b. We consider $check(\text{array}, i, \omega, b)$ to be a *sentinel check* of array when control flow exits the loop if b is true. Thinking in terms of a dynamic execution of the loop, the loop contains a *mandatory sentinel check* when every execution from L_{entry} looping back to L_{entry} contains at least one sentinel check.

We determine whether a sentinel check of array is mandatory using a depth-first search through the loop body. If at least one sentinel check of array must execute on every possible iteration of the loop, then this loop treats array as sentinel-terminated. Per key assumption 2, if any loop treats array as sentinel-terminated, then we annotate array as sentinel-terminated.

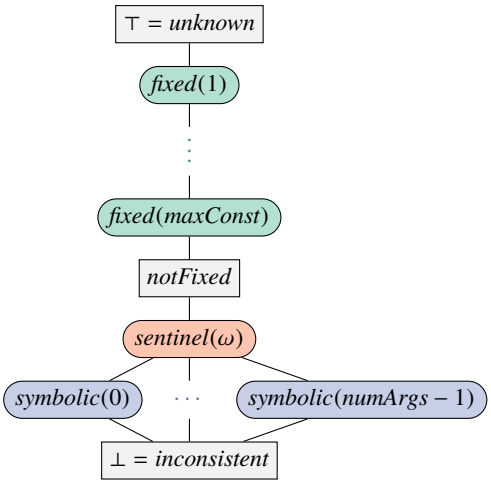

Figure 1: Result lattice for any single array under analysis in a function with *numArgs* arguments and no constant index larger than *maxConst*. Ellipses notwithstanding, the lattice is finite in both width and height.

Notice a deviation from our formal definition of a sentinel-terminated array in Section 4.1. Even with a mandatory sentinel check in a loop, NUL characters may be skipped over in the course of an iteration. The loop counter could increment by some value other than 1, or reads and writes outside the loop may occur. In this case, theoretically, sentinel characters might be passed over. Due to key assumption 3, we ignore this possibility in order to arrive at a more complete approach. Such an assumption would be unacceptable if we were attempting to check the memory safety property.

4.6 Merging Length Types

To this point, we have discussed how to approximate whether an array has each of the formal length properties within a single function. Our formal definition of length types in Section 4.1 technically allows for any combination of the three length types. However, our goal is to produce source-code level annotations that facilitate cross-language bindings. We are interested in length properties that rely more on developer intent than on the physical layout of the arrays in memory. For this reason, and due to key assumption 1 (see Section 4.2), we produce at most one annotation per argument, even if more than one could apply. Our goal is to provide the most helpful language bindings possible, so we make an effort to produce the most helpful annotation consistent with the analysis. Although multiple length types may be correct, we attempt to determine the most general one, based on the particular domain of language bindings for C libraries.

We also extend our analysis beyond individual procedures. In order to address internal calls from one library function to another, we iterate until we reach a fixed point. As we iterate, we combine results from different parts of our analysis to select at most one annotation per array argument. We endeavor to select the most general (still correct) annotation for each array argument. Figure 1 compactly summarizes our scheme for determining which annotations are the most general. The most general annotation is considered to be the greatest lower bound of this lattice. Our analysis is guaranteed to terminate, because we only replace an annotation with a more general annotation and there are a finite number of possible annotations.

Let $\textit{fixed}(n)$ denote a fixed-length type where n is the fixed length of the array. Similarly, $\textit{symbolic}(n)$ represents a symbolic-length type where n is the argument number of the argument representing the length. The sentinel-terminated type with sentinel value NUL is represented as $\textit{sentinel}(\omega)$.

We also use three special length types: $\boxed{\textit{unknown}}$, $\boxed{\textit{notFixed}}$, and $\boxed{\textit{inconsistent}}$. These types do not correspond to annotations, but represent arrays with intermediate types. $\boxed{\textit{unknown}}$ means that the argument is compatible with any length type. Often, this means that there are no accesses to elements from the array at all. $\boxed{\textit{notFixed}}$ means that the argument is compatible with any length type except $\textit{fixed}(n)$ for any n. This can happen when any non-constant index into the array is present. Often these non-constant indices are also symbolic length, but it is also possible that the length of the array is determined in some other way. $\boxed{\textit{inconsistent}}$ means that multiple, incompatible length types appear to be present. For example, the array might be accessed up to locations n and m, which both are additional arguments to the function. In this event, the most general annotation we can provide is no annotation at all, since neither piece of length information is truly safe to present. The lattice is most general at the top, which is consistent with any length type; it is least general at the bottom, which is consistent with no length type.

For purposes of producing annotations, it is most useful to present only the single most general length type. Our notion of generality is the one that selects the single binding that exposes the most functionality. When multiple length types are present, we take the meet ($\sqcap$) in the lattice depicted in Figure 1. For example, $\textit{fixed}(n) \sqcap \textit{fixed}(m)$ yields $\textit{fixed}(\max(n,m))$. In general, fixed-length types defer to any other kind of length: an array that is treated as both fixed-length and sentinel-terminated is assumed to be sentinel-terminated; an array that is treated as having both fixed and symbolic lengths is deemed to have symbolic length overall. We choose this to be the most general because a fixed-length array can always be used where a sentinel-terminated or symbolic-length array is used. The only requirement is that it be sentinel-terminated or the length be passed as an argument as appropriate, and the binding can hide this work. Symbolic lengths also subsume sentinels: for all n, $\textit{sentinel}(\omega) \sqcap \textit{symbolic}(n) = \textit{symbolic}(n)$. We consider symbolic length to be more general because a sentinel-terminated array has a length that might be passed as the symbolic-length. On the other hand, a symbolic-length array need not end with a terminating sentinel character, and worse, might contain the sentinel character well before the logical end. The binding would need to determine how to handle this, and may make the wrong decision. Finally, mismatched symbolic lengths are incompatible: $\textit{symbolic}(n) \sqcap \textit{symbolic}(m) = \boxed{\textit{inconsistent}}$ unless $n = m$. In this event, we have no recourse: there is no way to determine which of the two symbolic lengths were intended by the developer, and any assumption may cause a confusing binding to be created.

4.7 Structure Information

C structures can contain pointers which have the same ambiguities as argument pointers, and can be analyzed similarly. Like C pointer arguments, structure elements that are pointers also require their length to be stored implicitly. This can be done in the form of an additional structure field representing the length, or the structure field can be sentinel-terminated, or may have a fixed, known size. A structure field has a length property if any instance of the structure treats the field that way, as per key assumption 2. Once we have determined a length property for the structure field, consider that any array arguments stored in it could be accessed wherever the structure field is accessed. Thus, if a structure field element is ever treated as fixed-length, sentinel-terminated or symbolic-length, then any pointer arguments stored into that field must follow the same length idiom (key assumption 1). This potentially allows us to retrieve length information about array arguments that would otherwise be

impossible to determine, for example, in a setter function taking a pointer to a structure along with the data to store in the structure. After determining the length properties of structure fields (as in Section 4), we search for store operations that store an array (either an argument or structure field) into an annotated structure field. We then propagate this length information to the stored array as well. In theory, this also gives us more information about the structure arguments to functions, as well. GObject-Introspection currently does not support such annotations on structures, presumably because most functions that require a structure are not part of the external API that would require a binding. If structure annotations are available in the future, our tool should provide these with minimal modification.

We implemented such an analysis and combined it with our argument analyses. This did allow us to recover length information for a handful of array arguments that had previously not been recovered, but it also slowed execution time massively. Further details on these results can be found in Section 5.5. This overhead is likely because of the vast number of stores into structure elements in a sizable library. Further, it seems that many of the functions benefiting from this new information were not part of the public API of the library, meaning that the end user will not benefit from annotating these arguments in the first place.

4.8 Notes on Soundness and Completeness

Most length analyses, particularly those for verifying the memory safety property, attempt to be sound or complete. Soundness requires never erroneously reporting a length type; completeness requires reporting all length types. We sacrifice both soundness and completeness in favor of practical utility.

4.8.1 Practical Trade-Offs for Useful Bindings

We find a trade-off between two competing concerns. On the one hand, finding the physical lengths in memory of each array argument produces more useful bindings than finding the highest-numbered element a function will access. However, this often cannot be statically determined without introducing unsoundness. On the other hand, finding the maximum array offset is more frequently statically discoverable. However, this can produce less useful bindings, since the last used element may or may not correspond to the allocated length of the array.

A sound analysis would necessarily miss cases where arrays seem to have different types of lengths in different contexts. We assume this is the result of analysis imprecision, rather than a violation of key assumption 1: library writers treat arrays as though they have only a single type of length. This allows us to report length annotations where a sound analysis could not. A complete analysis, on the other hand, would necessarily retrieve some incorrect length information, which would produce incorrect bindings. We strive to avoid producing incorrect annotations, and so cannot take a complete approach, either.

Therefore, we take an approach which is neither sound nor complete, and thus we may produce both false positives and false negatives. Recall from Section 4.6 that each array has a most general correct annotation. Consider a *false positive* to be a function argument annotation identifying an "incorrect" length: i.e., any annotation but the most general correct one. A *false negative* fails to attribute the correct annotation to a function argument that requires an annotation. Note that a single annotation can be both a false positive and a false negative if it identifies an incorrect annotation in place of the correct one. For example, if an array has length $\mathit{fixed}(8)$, reporting length $\mathit{fixed}(7)$ both identifies an incorrect annotation and fails to identify a correct annotation. In one sense, this is the harshest method of assessing false positives and false negatives that we could use. Not only do we fail to award "partial credit" for inferring true length properties that are less general, but this incurs a false positive in addition to a false negative.

4.8.2 Sources of False Positives and False Negatives

Even with an unsound and incomplete approach, it is important to clearly identify the kinds and causes of potential errors, and to mitigate these risks as much as is practical. We find both false positives and false negatives resulting from our tool, but false positives are much less common. We intentionally designed our tool to produce more false negatives than false positives: a false negative creates a binding that is less idiomatic but still usable, whereas a false positive can render an API unusable. Such annotations may, for example, unnecessarily hide arguments or overly restrict their types.

Our analysis is subject to imprecision resulting from pointer aliasing, as we do not perform an alias analysis. This could result in a false positive if the array has inconsistent types across aliases (violating key assumption 1). For example, if the first $\mathsf{y}-1$ elements are accessed from array, and the y^{th} element is accessed only through an alias of array, we would incorrectly report that array has symbolic length y. However, in practice, we have not seen such false positives, and believe this to be unusual. Thus, the additional time and space overhead required for alias detection is not merited. Aliasing also could theoretically result in false negatives, if elements from array are accessed only via an alias. In that case, we will report that no length information is available for array. This, also, has proved to be rare in practice.

We may incur false negatives in the presence of variadic functions, which do not accept a fixed number of arguments. The length of a variadic list of arguments is usually determined by a format string. While many arguments passed into variadic functions like printf are strings, symbolic-length arrays, or fixed-length arrays, many are not. It is possible to identify variadic arguments, but it is not possible to determine their types without examining the format string, which contains more complex type information than we support. Variadic arguments serve as an extreme example of treating arguments as having different types depending on context, so key assumptions 1 and 2 do not apply.

Our last causes of false positives and negatives arise from external sources. SRA itself is unsound and incomplete, which may cause us to produce false positives and negatives. Our approach can also be incomplete if the only evidence for an argument's length is in a call to an external library function, whose code is not available to analyze. In that case, we allow the user to provide as input a set of hand-created annotations.

5. EXPERIMENTAL EVALUATION

We have implemented the analyses described in Section 4 using LLVM 3.7 [12]. Our implementation focuses on the special case where the sentinel value is a zero of any type, as this is the standard way to represent C strings. This is motivated by our target for language bindings, GObject-Introspection, which currently only has annotation-level support for zero-terminated arrays. Our tool operates on LLVM bitcode, and therefore is easily incorporated into any Clang-compatible build or analysis tool chain. All experiments were run on one 2.67 GHz CPU of a desktop workstation with 24 GB of RAM running Red Hat Enterprise Linux 7.

5.1 Test Subject Selection

We have evaluated our tool on the following libraries:

- gck v3.18 implements PKCS #11, a form of public key cryptography [6].

Table 1: Library details. KLoC measures thousands of lines of source code, estimated with SLOCCount [23].

			Number of Function Arguments				
Name	KLoC	Number of Functions	All	Symbolic	Fixed	Sentinel	Analysis Time (sec)
glib	151	1,813	4,092	77	12	483	211
gio	188	4,948	11,506	66	11	1,052	286
gck	15	247	719	26	0	12	7
telepathy-glib	151	917	2,016	25	0	155	275
libgit2	151	3,668	8,750	89	16	948	151
libssh2	39	349	1,266	125	6	81	14

Table 2: Rates of correct and incorrect analysis results in complete libraries

	Rate of True Positives			Rate of False Positives		
Name	Symbolic	Fixed	Sentinel	Symbolic	Fixed	Sentinel
glib	0.7143	0.9167	0.8903	0.0002	0.0005	0.0044
gio	0.8030	0.7273	0.7814	0.0000	0.0000	0.0061
libgit2	0.7191	0.8125	0.8565	0.0001	0.0006	0.0003
libssh2	0.7280	0.8333	0.8765	0.0000	0.0047	0.0000
Arithmetic Mean	0.7411	0.8224	0.8512	0.0001	0.0015	0.0027

- gio v2.46.2 is a virtual file systems API [7].
- glib v2.46.2 provides a framework for C libraries, including utility functions and a **struct**-based object system [8].
- libgit2 v0.23.4 implements the Git core methods as a linkable library [14].
- libssh2 v1.6.1 is an implementation of the SSH2 protocol in an extensible C framework [10].
- telepathy-glib v0.23.3 is a D-Bus framework for real-time communication [22].

Most of these libraries are part of the GNOME Project [5] and already have GObject-Introspection annotations, authored by the library writers. The notable exceptions are libgit2 and libssh2, neither of which is a GNOME library. The version of libgit2 we analyzed had no GObject-Introspection annotations, though annotations appeared in a later release. libssh2 has no GObject-Introspection annotations as of this writing, and no language bindings to our knowledge. These libraries assess our technique on code that was not specifically written with these annotations in mind. Table 1 provides more details on our test subjects. To determine ground truth on the number of symbolic-length, fixed-length, and sentinel-terminated arrays, we manually inspected each library. Identifying whether a C argument is an array or a pointer is a difficult task in its own right, and not one we attempt. We therefore do not report the number of array arguments in each function. All arguments that our tool infers to possess a length property must be arrays: however, some arrays may not follow any of the length types we detect.

For each library, we made a reasonable attempt to identify and analyze any dependencies, whether manually or with the help of our tool. We were unable to analyze most of libc, due to many important functions being implemented in assembly rather than C. Therefore, we manually selected several such important functions and annotated them by hand. When analyzing each library, we passed along information for all of its dependencies as generated by our tool, and additionally included our hand-crafted annotations for libc. Thus, some dependency information may be incorrect where our tool is imprecise. We note the true positive rate and false positive rate of each type of length information in Tables 2 and 3. The true positive rate measures the ratio of correct annotations produced to correct (and most general) annotations, whether produced or not. The false positive rate measures the ratio of incorrect annotations produced to arguments which should not have that annotation. A higher true positive rate and lower false positive rate is desirable, though we prioritize a lower false positive rate per Section 4.8.2.

5.2 Full Annotation Results

For gio, glib, libgit2, and libssh2, we manually annotated the full library to use as a baseline for comparison. This provides a complete picture of how many sentinel-terminated, symbolic-length, and fixed-length arrays are in the libraries, but costs significant time. Indeed, our experience indicates that libraries comparable in size to these take upwards of eight hours to manually annotate. Table 2 summarizes our findings. Over each of the full libraries, we achieve a minimum true positive rate of 0.7 for each type of length property, indicating that we produce at least 70% of the correct annotations. Every false positive for sentinel-terminated arrays belongs to a class of problems discussed further in Section 5.4. In brief, each arises from a function that accepts an array argument and a length argument, but treats the array as NUL-terminated if the length is negative. Recall from Section 4.4 that we could introduce false positives if a library writer ever mixes constant and symbolic indexes in a fixed-length array. However, only one library in our suite contains such code, libssh2, and manual inspection verifies the two occasions when this occurs as truly symbolic-length arrays.

Our results for libssh2 are of particular interest. libssh2 appears to have been built without awareness of GObject-Introspection annotations. We see no evidence that polyglot interoperability was factored into this library's design in any way. Yet our analysis performs about as well here as on the other (hand-annotated) libraries. Furthermore, our true positive rate for sentinel-terminated arrays is significantly higher than for most other libraries. These results indicate that our tool can support even libraries that were not built with language bindings in mind. Our approach can help developers retrofit GObject-Introspection bindings onto existing libraries without requiring analysis-friendly design from the start.

Table 3: Rates of correct and incorrect analysis results in external library APIs. gck and telepathy-glib use no fixed-length arrays.

	Rate of True Positives			Rate of False Positives		
Name	Symbolic	Fixed	Sentinel	Symbolic	Fixed	Sentinel
gck	0.8077	—	1.0000	0.0000	—	0.0014
glib	0.7222	1.0000	0.8904	0.0004	0.0000	0.0056
gio	0.6875	0.5000	0.7051	0.0000	0.0000	0.0010
telepathy-glib	0.8800	—	0.6839	0.0000	—	0.0032
telepathy-glib + hints	0.8800	—	0.7806	0.0000	—	0.0037
Arithmetic Mean	0.7955	0.7500	0.8120	0.0001	0.0000	0.0030

5.3 API Results

For libraries with existing GObject-Introspection language bindings (gio and glib) and the remaining libraries (gck and telepathy-glib), we examine the annotations already present in the source code. These are produced only for the subset of the library intended to be exposed to the end user in the form of an API. We manually examined those functions in the API where our tool produced a different annotation than the human did. This method of determining ground truth is less precise than manually determining the correct annotations for every function argument in the library, but consumes much less time, and allows us to determine how well we perform on the parts of a library that ultimately require the annotations: the API.

5.3.1 Automated Analysis

Table 3 shows our results on these libraries. For the most part, our automated approach does quite well. Rates for symbolic lengths generally improve upon those for complete libraries in Table 2. The notable exception is the gio library, which exposes only eight symbolic-length arrays in its public API. Our 67% true positive rate for sentinel-terminated arrays in telepathy-glib is likewise anomalous. This is due to heavy use of variadic functions within telepathy-glib. As discussed in Section 4.8.2, variadic functions pose a problem for our analysis, as the type information may be dependent on the content of a format string. Our tool has no way of reasoning about how to extract type information from format strings. Thus, we are unable to annotate any arguments whose type information is discoverable solely through the use of variadic functions. This accounts for most incorrectly-analyzed sentinel-terminated arguments in telepathy-glib.

Note that a developer could provide a set of manually annotated functions to recover from this situation. When we manually annotated ten functions that call variadic functions, the results improved significantly (see the "telepathy-glib + hints" row of Table 3). More manual annotations could provide a larger benefit, but even this amount improves upon our results. GObject-Introspection annotations are based on fixed argument positions, and cannot support annotations of variadic arguments, but the library author could annotate common functions that make calls to variadic functions.

5.3.2 Human Errors

We motivated this work with the claim that creating bindings manually is tedious and error-prone. The numerous mistakes we found in human-authored annotations support this claim. Humans' errors are qualitatively different from those produced by our tool. Understanding this mismatch helps illustrate how our approach can complement human efforts.

Errors by human annotators seem to stem from inattention or misinterpretation of functions in dependencies. One such mistake is reporting that an argument has a symbolic length when it is only ever passed to a function call in some dependency, which does not treat the argument this way. This happens especially commonly when the names of the arguments are misleadingly suggestive of a symbolic length relationship between two arguments. These mistakes occur even when the documentation does not suggest that the writers of these dependencies considered them to be symbolic-length. In these cases, it is likely that the library writer did not find it worth the time to track down the source-level annotations in each dependency to see what the length type of the argument actually is. Rather, they relied on the name of the argument. Quantitatively, we can see in Table 4 that humans perform marginally better overall, but have a higher rate of false positives for both symbolic-length and sentinel-terminated arrays. Humans take much more time to produce these annotations, while our analysis runs in under five minutes on each library we considered. One bug that we detected in the library gio was fixed since the time we ran our analysis. We have submitted a bug report for the remainder of the human errors our analysis detected in the library gio, which is awaiting action by the developers.[1]

```
static inline gboolean
contains_non_ascii (const gchar *str, gint len)
{
    const gchar *p;

    for (p = str; len == -1 ? *p : p < str + len; p++) {
        if ((guchar)*p > 0x80)
            return TRUE;
    }
    return FALSE;
}
```

Listing 6: Real-world function with inconsistent treatment of an argument's length, taken from glib

5.4 Empirical False Positives

Most of our false positives arise when an array exhibits multiple length properties (violating key assumption 1), particularly in the symbolic-length case. For example, in Listing 6, string is treated as sentinel-terminated by NUL when len is −1, and as having symbolic length len otherwise. This appears quite often in real-world code, evidently for efficiency; if the caller already knows the string's length, it can pass that down to avoid recomputing it in the library. Technically, these functions can be used by character arrays that are not strings as well, such as arrays with embedded NULs. By producing a binding that only accepts strings, we remove functionality. Because our analysis is not path sensitive, we are unable to identify that string is treated as NUL-terminated only under some circumstances. We see that string is treated as sentinel-terminated when it is used in the call

[1] https://bugzilla.gnome.org/show_bug.cgi?id=765063

Table 4: Rates of correct and incorrect human-authored annotations in external library APIs

	Rate of True Positives			Rate of False Positives		
Name	Symbolic	Fixed	Sentinel	Symbolic	Fixed	Sentinel
gck	0.8846	—	1.0000	0.0252	—	0.0028
telepathy-glib	0.8800	—	0.9484	0.0045	—	0.0620
Arithmetic Mean	0.8823	—	0.9742	0.0148	—	0.0324

Table 5: Change in sentinel-terminated argument counts after adding structure information analysis

Name	Analysis Time (min)	New True Positives	New False Positives
gio	>5,760	—	—
glib	508	0	97
libgit2	2,376	5	10
libssh2	7	3	3

to strlen, and infer that it must be sentinel-terminated, although the strlen call is conditional on the value of len. We chose to combine analysis results using our Hasse diagram in Figure 1 in order to combat this issue. We eliminate many of these false positives by combining our sentinel-terminated and symbolic-length analyses, and produce only the more general symbolic-length annotation. We are unable to eliminate these false positives only when our analysis fails to detect that the array may have a symbolic length.

5.5 Structure Information Results

We extended our implementation to analyze structure fields as described in Section 4.7. We only found different results in the sentinel-terminated case, so we just report these results. Table 5 shows that this does improve some cases, allowing us to discover a few more sentinel-terminated arrays. However, the sheer number of function arguments to annotate causes the global impact of these improvements to be quite modest, and we introduce new false positives as a result of structure fields being treated inconsistently across several functions. There do not appear to be many arrays whose length information could be recovered by examining structures.

Further, the analysis now has far more work to do, making performance a serious concern. We were unable to completely analyze all of the libraries in Section 5.1 due to time constraints and dependency information. Most of the GNOME libraries depend on glib and gio. gio analysis timed out after four days. Therefore, we could not analyze any libraries dependent on gio and obtain comparable results to our other experiments. While further performance tuning of our implementation is possible, the results (see Table 5) suggest that the marginal benefits may not make structure analysis worthwhile.

6. FUTURE WORK

While our current approach substantially reduces the manual work load of generating high-quality bindings, further improvements are possible. One possible future direction is to consider any client code that may be available. This would be optional input that would allow the user to supply representative client code that uses the library. One source of the client code might be the library itself, which often call into its own public API. This analysis would be substantially different from the one described here, as it could take allocation points into account in the style of SALInfer [11].

We could also expand our analyses to handle function pointers. Function pointers may be passed as callbacks into a C function, and the length idiom used by array arguments to the function may be partially or completely dependent on the definition of the callback. In order for to analyze such functions, we would need to analyze all the callbacks passed to such functions within a library.

We have been using GObject-Introspection annotations as our ultimate analysis target. This ensures that our analysis findings can be put to good use, but also limits how much detail we try to recover. We could track other kinds of length information, such as determining when a function accepts a start pointer and end pointer. We could also infer *predicated type* information, which determines the length information of a particular argument given the values of other arguments. For example, a predicated description of string from Listing 6 would state that it is sentinel-terminated by NUL if len is -1, or has symbolic length len otherwise. GObject-Introspection can neither express nor use array lengths such as these, but if they are common enough in practice, that may justify extending GObject-Introspection to include them as well.

7. CONCLUSIONS

We have presented a system for automatically inferring developer intent about array argument lengths. This task bears some similarity to that of checking that all array accesses are memory safe. However, our focus on language bindings mandates a different design, tuned to allow different kinds of imprecision and to use heuristics that would be unacceptable when checking for memory safety violations. Instead of finding mistakes, we are looking for trends in the kind of length the library developer expects.

Empirical evaluation shows that we produce significantly fewer false positives than existing hand-written annotations. Our results also indicate that our tool performs well even with libraries that were not built with the goal of being accessible to other languages.

The challenge of producing high-quality bindings is large. Our inferred array lengths provide an important piece of that larger puzzle. In cooperation with prior work by others, these analyses begin to form a comprehensive suite that substantially reduces the manual effort needed to cross language boundaries. In so doing, we liberate polyglot programmers to mix and use the best tools, languages, and libraries available.

8. ACKNOWLEDGMENTS

This research was supported in part by grants from CAPEs and CNPq; DARPA MUSE award FA8750-14-2-0270; and NSF grants CCF-0953478, CCF-1217582, CCF-1318489, and CCF-1420866. Opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the sponsoring agencies. We would like to thank Peter Ohmann, for insightful comments and contributions to the ideas in this paper.

9. REFERENCES

[1] P. Alves, F. Gruber, J. Doerfert, A. Lamprineas, T. Grosser, F. Rastello, and F. M. Q. a. Pereira. Runtime pointer disambiguation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 589–606, New York, NY, USA, 2015. ACM. . URL http://doi.acm.org/10.1145/2814270.2814285.

[2] D. M. Beazley. Swig: An easy to use tool for integrating scripting languages with c and c++. In *Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4*, TCLTK'96, pages 15–15, Berkeley, CA, USA, 1996. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267498.1267513.

[3] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, LCTES '03, pages 69–80, New York, NY, USA, 2003. ACM. . URL http://doi.acm.org/10.1145/780732.780743.

[4] M. Furr and J. S. Foster. Checking type safety of foreign function calls. *ACM Trans. Program. Lang. Syst.*, 30(4): 18:1–18:63, Aug. 2008. . URL http://doi.acm.org/10.1145/1377492.1377493.

[5] The GNOME Project. GNOME, Nov. 2015. URL https://www.gnome.org/.

[6] The GNOME Project. Gck library reference manual, Oct. 2015. URL https://developer.gnome.org/gck/3.18/.

[7] The GNOME Project. GIO reference manual, Nov. 2015. URL https://developer.gnome.org/gio/2.46/.

[8] The GNOME Project. GLib reference manual, Nov. 2015. URL https://developer.gnome.org/glib/2.46/.

[9] The GNOME Project. GObject-Introspection Annotations, June 2015. URL https://wiki.gnome.org/Projects/GObjectIntrospection/Annotations.

[10] S. Golemon, M. Gusarov, The Written Word, Inc., E. Fant, D. Stenberg, and S. Josefsson. libssh2, Oct. 2015. URL http://www.libssh2.org/.

[11] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 232–241, New York, NY, USA, 2006. ACM. . URL http://doi.acm.org/10.1145/1134285.1134319.

[12] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. URL http://dl.acm.org/citation.cfm?id=977395.977673.

[13] W. Le and M. L. Soffa. Refining buffer overflow detection via demand-driven path-sensitive analysis. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, pages 63–68, New York, NY, USA, 2007. ACM. . URL http://doi.acm.org/10.1145/1251535.1251546.

[14] The libgit2 contributors. libgit2, Oct. 2015. URL https://libgit2.github.com/.

[15] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. Muvi: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. *SIGOPS Oper. Syst. Rev.*, 41(6):103–116, Oct. 2007. . URL http://doi.acm.org/10.1145/1323293.1294272.

[16] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 245–258, New York, NY, USA, 2009. ACM. . URL http://doi.acm.org/10.1145/1542476.1542504.

[17] H. Nazaré, I. Maffra, W. Santos, L. Barbosa, L. Gonnord, and F. M. Quintão Pereira. Validation of memory accesses through symbolic analyses. *SIGPLAN Not.*, 49(10):791–809, Oct. 2014. . URL http://doi.acm.org/10.1145/2714064.2660205.

[18] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, May 2005. . URL http://doi.acm.org/10.1145/1065887.1065892.

[19] T. Ravitch and B. Liblit. Analyzing memory ownership patterns in C libraries. In P. Cheng and E. Petrank, editors, *International Symposium on Memory Management, ISMM '13, Seattle, WA, USA - June 20 - 20, 2013*, pages 97–108. ACM, 2013. . URL http://doi.acm.org/10.1145/2464157.2464162.

[20] T. Ravitch, S. Jackson, E. Aderhold, and B. Liblit. Automatic generation of library bindings using static analysis. In M. Hind and A. Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 352–362. ACM, 2009. . URL http://doi.acm.org/10.1145/1542476.1542516.

[21] R. Rugina and M. C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM Trans. Program. Lang. Syst.*, 27(2):185–235, Mar. 2005. . URL http://doi.acm.org/10.1145/1057387.1057388.

[22] The Telepathy Project. telepathy, Aug. 2014. URL http://telepathy.freedesktop.org/.

[23] D. A. Wheeler. SLOCCount, June 2015. URL http://www.dwheeler.com/sloccount/.

[24] T. Wies, V. Kuncak, K. Zee, A. Podelski, M. Rinard, T. Wies, V. Kuncak, K. Zee, A. Podelski, and M. Rinard. Verifying complex properties using symbolic shape analysis. In *In Workshop on heap abstraction and verification*, 2007.