Static Analysis Support for Domains With Limited Type Information

By

Alisa J. Maas

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

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2021

Date of final oral examination: 06/25/2021

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

Ben Liblit, former Associate Professor, Computer Sciences
Tom Reps, Professor, Computer Sciences
Loris D'Antoni, Assistant Professor, Computer Sciences
Julian Dolby, Principal Research Staff Member, IBM
Dimitris Papailiopoulos, Associate Professor, Electrical & Computer Engineering

To frustrated developers everywhere.

ACKNOWLEDGMENTS

People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on each other, like a wall of mini stones.

— Donald Knuth

There are so many people who have been a part of helping me succeed, and it is fitting that the first traditional section of a dissertation acknowledges their support and encouragement. I could not be who I am without all of you, even the ones I haven't met.

I would like to thank my advisor, Ben Liblit, for everything. His curiosity and enthusiasm was a constant source of inspiration for me. Whenever I walked into a meeting feeling discouraged about my progress, Ben helped me to recognize the parts of my contributions that I was dismissive about. I seldom left meetings feeling anything other than excited to get back to work, and he instilled in me a desire to get things done *right*, not just done. Words are inadequate to express how grateful I am for his guidance.

Julian Dolby served as a surrogate advisor when Ben transitioned to a full-time industry position at Amazon, and I am very thankful for his time, advice, and support, particularly at the end of my PhD. He was ever-more optimistic about research directions than I am inclined to be, and I know my research would be far worse off without his encouragement to test ideas rather than discarding them when my skept-ometer was off the charts. I would also like to thank the rest of my committee: Tom Reps, Loris D'Antoni, and Dimitris Papiliopoulos. Their feedback has helped me to focus my research interests in fruitful directions and their support has enabled me to push on when progress was challenging.

The University of Wisconsin–Madison's faculty, staff, and resources, both Computer Sciences and the school at large, made this research possible. I want to particularly thank Angela Thorp, the Graduate Program Coordinator. Angela provided invaluable support during my PhD process, far above and beyond the role of a program coordinator. She was supportive, empathetic, and offered a vital combination of encouragement and practical advice. Her hard work makes the logistics of navigating the PhD process possible.

I also would like to particularly thank the Center for High Throughput Computing (CHTC) at the University of Wisconsin–Madison. The team there was incredibly helpful when I needed to run distributed experiments. I mainly interacted with Christina Koch and Jess Vera, both of whom provided troubleshooting and advice for how to streamline my experiments in a way that interacted well with the CHTC's interface.

Discussions with other members of the Liblit research group were very formative for me, and I would like to thank Peter Ohmann, Dave Bingham Brown, Pallavi Ghosh, Zi Wang, and Naveen Neelakandan, who were subject to group meetings with me. I inflicted each with numerous practice talks, brainstorming sessions, and endless questions. I am forever grateful for their support, particularly Peter, my former officemate. Although I did not overlap with him during my graduate studies, I am also very grateful for Tristan Ravitch's research, as much of my own work was directly inspired by his work on polyglot programming during his time in the Liblit group.

Although neither of them has ever taken a class in computer science, my parents (Aimee & David Maas) were formative for my growth into a burgeoning scientist and researcher. From a young age, they encouraged my questions and treated my thoughts and opinions seriously. My mom's patience with us and her ability to remain calm even during stressful situations inspires me. She instilled in me her ability to break a stressful task down into smaller, manageable steps, without which I would never have completed any research projects. When I wanted to be an English teacher, my dad took it seriously, and made sure my writing skills could handle such a possible future career. He also first put into my head the idea that I might enjoy computer science. Without his suggesting it, I probably never would have even considered taking that first class.

My sisters and their families are an integral part of my life, and I cannot imagine having done this without them. To Abby Andrews, her husband Jacob, and my nephew Ivan and niece Evelyn; and Kayla Kirkland, her husband Joseph, and my nephews Liam and Levi: thank you. I am incredibly grateful to have grown up with Abby and Kayla. Abby's creativeness and humor and love is irreplaceable, as is Kayla's resilience, love, and strength. I could not have picked better sisters to grow up with. I have been so thankful to welcome into the family both brothers-in-law and my niece and nephews. I want to thank the rest of my extended family: grandparents, aunts, uncles, cousins, cousins-of-cousins and others. You have been there for me all my life. I have no words to express how much I love and appreciate you all. Thank you.

I would never have taken a second computer science class if it were not for my first programming teacher, Thomas VanDrunen, whose enthusiasm for computer science helped me realize how different programming was from the boring, clinical skill I expected. The other computer science students at my undergraduate school, Wheaton College, provided a tightly-knit community and introduced me to so many others who were passionate about Making Cool Things. In particular I want to thank Lily & James Hampton, John Charles & Emily Bauschatz, Drew & Suzanne Hannay, Kendall Park, Dan Opdyke, Andrew Wolfe,

Leanne Miller, Becca Meloy, Tim MacDonald, and Cheney Hester. And Duckjstra. Not all of them were officially computer science majors, but all of them made my time in the computer science department far more fun. Long live the quote wall!

My Madison friends also deserve many thanks for their love, support, and encouragement. They have played board games with me, stood in riddle-solving circles, and invented creative games to play far too late at night. Simply listing each of your names does not feel like enough. I would not have remained sane without you all. From the CS lunch/games crew, I especially want to thank Alex Cobian, for his endless support and treasure chest of board games; Alexi Brooks (and partner, Beth Miller) for hosting many game nights and for being a valued friend; Steven Brown, for more inside jokes than I can count ("We're missing pieces!") and many outings to cornfields and pumpkin patches; and so many others (Corey Schulz, Matthew Bernstein, all the Kevins except Thursday Kevin, Monica Thompson & Austen Ott, Chris Feilbach, and Mark Mansi). And, of course, Sally Seacow, the indomitable guardian of the CS 302 Testing Server.

My church family, Laketrails, has supported and encouraged me more than they could possibly know. Learning and growing with them has helped me to be a better and happier human. I particularly want to thank my community group members (Max & Ann Harris, Claire & Nelson Mankey, Carl & Diana Anderson, Carla Winsor, Jeremy Erikson, Tucker Meyers, Josh Land, and Annette Dean) for the way they have loved me, prayed with and for me, and helped shape me into who I am today.

So many friends do not fit neatly in a category, but I am no less thankful for them. Some are besties, conference-friends or cross-country friendships, and some defy categorization. Thank you so much, Katie Pinson, Ming-Ho Yee, Jake Hughes, Daniel Pelsmaeker, Jaime & Dave Ballard, and Kira Lawrence. You have served as friends, emotional support, and been bright spots in otherwise stressful times. Your presence in my life has vastly improved it.

Although I have never met any of them, I am incredibly grateful for the authors whose books have made an impact on me. Some (though certainly not all!) of these authors are: Patricia C. Wrede, Diana Wynne Jones, Patrick Rothfuss, Brandon Sanderson, Timothy Zahn, Sherwood Smith, Naomi Novik, Madeline L'Engle, Vivian Vande Velde, C. S. Lewis, C. Dale Brittain, Gail Carson Levine, Lois Lowry, Rinsai Rossetti, Intisar Khanani, Drew Hayes, Jacyln Moriarty, Ron Stallworth, Carolyn Keene, and Marshall Ryan Maresca. Some books give me a flash of insight into the incredibly diverse ways people operate, some help me to feel seen and loved, and some just give me the space to slip into another world for a few hours. I am a better human for having read books by these authors. They instill me with curiosity

and empathy, and for that, I will never stop being grateful. Much of my love for programming developed from a love of worldbuilding and realistic characters: in some ways, the power to construct a world where you define the rules belongs both to the writer and to the programmer.

This dissertation 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 author and do not necessarily reflect the views of the sponsoring agencies.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LISTINGS

ABSTRACT

When it comes to the task of writing a large-scale, real-world project, programmers rely on a host of tools: IDEs, debuggers, compiler optimizations, security checkers, test case generation and more. These tools are invaluable to allow developers to write quality code. However, tool support varies wildly dependent on which language the developer chooses to write in. Often, tool support is especially lacking in domains where the language itself makes some feature of the analysis more complicated or difficult. These domains pose new and interesting intellectual problems, but more importantly, they represent areas for improvement in developer tool support. This dissertation focuses on two domains where static analysis is complicated due to the shortcomings of the programming language itself.

First, we examine the domain of cross-language (or "polyglot") C programs, specifically programs that call C from a program written in another language. These programs frequently lack even the most basic of tool support at the cross-language boundaries, which ironically is where support is most needed. At cross-language borders, differences between the language performing the call (the "host" language) and the language the callee is implemented in (the "guest") language, can make constructing intuitive APIs challenging. We applied new techniques to identify developer *intent* as to how various arrays represent their length in program APIs. Our tool emits output that can be parsed by the GObjectIntrospection project[12], which automatically creates efficient, intuitive language bindings for C projects to a variety of host languages.

Second, we focus on the domain of JavaScript programs. JavaScript has been extremely popular in the last decade, holding the #1 mostly commonly used programming language for the past 8 years, according to the Stack Overflow Developer Survey in 2020 [32]. Its popularity looks likely to continue: in the same survey, it placed second in the list of languages developers most desired to learn. However, because of JavaScript's extremely dynamic type system, writing tools to support JavaScript developers is especially challenging. We identified one large bottleneck for JavaScript tool support, call-graphs, and applied new techniques to improve call-graph construction for JavaScript programs.

Part I

Introduction

1 INTRODUCTION

Ask ten programmers working on different tasks what their favorite language is, and you will get ten different answers. If the programmers are all in the same room, very likely a lively debate about the merits (. . . and pitfalls. . .) of each will ensue. Comics will be deployed, famous programmers will be quoted,[1] and participants will walk away with opinions more firmly held than when they arrived.

Despite deeply-entrenched opinions about their favorite languages, today's programmers typically use more than one programming language: according to the data from the StackOverflow Developer Survey in 2020 [32], developers reported using an average of 5.0 different languages in 2019, and reported that they would *like* to use 4.4 different languages on average in 2020.[2] Why do developers use so many languages when they care so deeply about which language they prefer? The answer is complicated. Some languages are simply more suited to particular tasks. In other situations, existing codebases ("legacy code") may require developers to either scrap previously completed-and-tested code and spend many dollars and man-hours re-implementing it, or work in an older language they may not prefer. At times, it can be a balancing act of the languages the team is fluent with, the tool support provided by each language, and the particular quirks and idioms that make a language more suited for one task or another.

This dissertation focuses on addressing the needs of developers particularly in need of tool support to accomplish their development aims. All too often, tools are written as proof-of-concept, and focus on the best-case scenarios, or languages most suited to analysis, rather than the languages that actually need the support the most. The challenging domains are very often the domains that most need tool support: where things are challenging to analyze automatically, generally they serve as fertile breeding grounds for bugs and bog down developer time and effort.

One particularly large hurdle to static analysis is domains where type information is extremely limited. Type information confounds static analysis in particular because, when present, it serves to constrain the space of possibilities. When not present, its lack interferes with basic needs program analysts have to provide adequate tool support. Its absence often causes developers difficulty as well. While writing in a language with a compiler that

[1] "The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offense."
—Edsger W. Dijkstra

[2] When we processed these data, we ignored reports from developers who did not list any languages from 2019.

complains over the slightest infraction has its own disadvantages, weak typing often leads to mistakes that can introduce bugs. We only wish we could address all such domains, but this dissertation at least addresses two – improved tool support for connecting modern languages to C, where the limited type information about pointers makes automatic bindings challenging; and JavaScript static analysis, where JavaScript's innately dynamic nature requires that all type information be constructed by an analysis. This dissertation, therefore, uses **static analysis driven by empirically-proven heuristics to recover missing type information and support traditional static analysis techniques.**

1.1 Why Bother With C?

The first part of this dissertation examines the domain of cross-language (or "polyglot") C programs, specifically programs that call C from a program written in another language. These programs frequently lack even the most basic of tool support at the cross-language boundaries, which ironically is where support is most needed. Most mono-language programmers take intuitive method calls for granted. However, at the cross-language borders, differences between the language performing the call (the "host" language) and the language the callee is implemented in (the "guest") language, can make constructing intuitive APIs challenging. Ideally, developers would work in whichever language is most convenient for the task at hand, the developer assigned to the task, and the development environment used. If needed, the developer could simply invoke functions written in legacy code. In the real world, this is not as simple as it sounds. Weaving together multiple programming languages into a single coherent program ("polyglot" development), can be very challenging depending on which languages the developer needs to use. Worse, there are a myriad of ways for extremely subtle bugs to creep into the program, hiding in unusual corners where the two languages interact in unexpected ways.

When it comes to unusual modes of interaction, older languages fall prey to this more often than newer languages, especially low-level languages like C. Data from Two Sigma Ventures [47] indicates that C appears more commonly used as a secondary development language, rather than the primary. (See Appendix A for more details on this data.) Imagine being a modern developer writing, say, a Python program. You come across a task that would be much more efficient with access to low-level data access that C excels at: in fact, you even find a C library that does more-or-less what you want. Unfortunately for you, it doesn't have an API that allows function calls from Python, only an API accessible from C code.

So you're faced with a decision: you can try to find a similar library in Python and take the performance hit, or you can attempt to stitch together your program and the C library. The second approach has one large problem: the representation of data types in C and Python are extremely different.

Therefore, to take this approach, you need to manually package up each argument as C expects. This is even more complicated than it sounds in practice: C represents data very differently from most modern programming languages, particularly as it does not have a concept of objects *per se*. C expects raw pointers to data, rather than a container object that may have multiple fields. Effectively, types that are represented as objects in object-oriented languages (and therefore have structured type hierarchies) are left without static typing and the safety it provides. One place where this difference proves especially stark is in the representation of arrays. Most modern languages treat arrays as container objects with indexing allowed, memory checks for safety if an index is out of range, and crucially, knowledge of the length of the array. C does not distinguish arrays from other pointers. It simply references a bit of memory, and the programmer can ask for an offset from that, with no guarantee that it will be valid, if they believe that bit of memory is conceptually part of the intended array. C developers are left to their own resources to devise conventions to track the logical and conceptual lengths of the data allocated, rather than having this baked into the type information provided with the array. Thus, copying a Python array to a new C array is more involved than simply allocating memory and copying over the data. You also have to determine how the function you plan to call will treat the array, and how it will decide on the length. Sometimes this is obvious from the names of the function arguments, but sometimes these names are unhelpful, even misleading. This problem inspired our approach in Chapter 2.

We focus particularly on the use case of host programs written in any language but C, not because it is easier to analyze[3], but because it is a place of especially great need for developers. Without array-length information, an automatic technique to extract cross-language bindings cannot correctly package up arrays and strings from high-level languages, instead requiring the developer to tediously package up any arrays or strings passed in, and manually copy out any that may have been changed or returned. We applied new techniques to identify developer *intent* as to how various arrays represent their length in program APIs, since program-level support for array lengths is unavailable. Our tool emits output that can be parsed by the

[3] "We choose to go to the Moon in this decade and do the other things, not because they are easy, but because they are hard."

—John F. Kennedy

GObjectIntrospection project[12], which automatically creates efficient, intuitive language bindings for C projects to a variety of host languages.

1.2 Aren't Call-Graphs a Solved Problem?

The second part of this dissertation focuses on the domain of JavaScript programs. Dynamically typed languages have stood the test of time, and continue to become more relevant and more important in modern software development. Stack Overflow's 2020 Developer Survey Results reports, 'Unsurprisingly, for the eighth year in a row, JavaScript has maintained it's stronghold as the most commonly used programming language." [32]. Furthermore, the 54% of respondents who reported using JavaScript in 2015 [31] rose to 68% by 2020 [32]. These Developer Survey Results also indicate that modern developers *want* to program in JavaScript: 18.5% of developers who do not currently use JavaScript indicated that they wanted to learn it in 2020. JavaScript is surpassed in this regard only by Python, another dynamically typed language, which 30% of developers desire to learn. Data from Two Sigma Ventures [47] agrees with this, as JavaScript features in 32 of the 100 most popular GitHub projects they list, and often occurs in very high proportion, indicating it commonly serves as a primary development language. Yet, despite this, Stack Overflow still finds 41.7% of developers in 2020 as "dreading" JavaScript: these are "developers who are currently using [JavaScript but] express no interest in continuing to do so."

In addition to concerns about developer contentedness, as dynamic languages (including JavaScript) grow ever-more-popular, so do significant security concerns and development costs for debugging. These concerns are widespread across dynamic languages, but especially relevant for code that manages secure user data, such as web development. Analysis tools to help developers find and correct security vulnerabilities can broadly be classified into dynamic and static strategies. Static approaches for program analysis reason over *all* possible executions of the program, while dynamic approaches focus on *specific* executions of the program. Dynamic strategies have a broad range of support in dynamically typed languages; however, the range of support for static program analysis techniques is generally far more limited. Yet if we wish to gain confidence that our programs are truly secure, an analysis reasoning about all executions of the program would be incredibly valuable.

Preventing developer dread and avoiding security vulnerabilities are difficult for any language, but are especially challenging for dynamically typed languages. One reason is the severe lack of static-analysis support for programs written in dynamically typed languages.

Without the tools they need to effectively debug and understand their program, developers may become frustrated and wish to switch to a language with better support. The lack of these tools can also easily lead to security vulnerabilities, which may only be found months or even years post-release. Yet dynamic languages are here to stay: since its inception in 1995, JavaScript has become solidly entrenched as an essential component of web development. It is not reasonable to expect developers to cease using dynamically typed languages, nor would it be beneficial to lose the many advantages of having dynamically typed languages at our disposal. Real code is messy, and static-analysis techniques must be robust enough to handle the realities of our development landscape, not only the most suited to easy analysis.

To that end, we focused on providing developers in dynamic languages the backbone of static-analysis techniques: call-graphs. Many static-analysis techniques rely heavily on a call-graph for information regarding interprocedural edges. In a statically-typed language, type information provides key insights that limits the number of possible callees at any given call site. In many dynamic languages, however, types are defined entirely dynamically. JavaScript, for instance, uses a prototype model rather than an object-model; JavaScript developers have no alternative but to develop their objects piecemeal, adding functions and fields to their objects on-the-fly. In effect, creating call-graphs becomes similar in nature to a points-to problem.

Our technique focuses on providing focused bursts of context sensitivity, based on a rigorous set of empirical tests, and seeks to do additional work upfront to avoid performing unnecessary work later. See Chapter 3 for details of our implementation. Equipping JavaScript developers with call-graphs makes it far easier to implement tools that rely on interprocedural information. Hopefully, this will better equip JavaScript and other developers in dynamic languages with the ability to write better code in a way that reduces developer dread and improves the development experience.

1.3 Impactfulness

This section is dedicated to the impacts that our work may have beyond the direct implications of the features added directly. While we contribute technical knowledge and novel analysis strategies, the impact of this research has the potential to more broadly impact computer science and related fields, and this section acknowledges this and lays out some of the anticipated effects. At a high level, of course, it is our hope that the tools we provide in this work will inspire more developers to produce tools for domains that are especially challenging

and also particularly in need of tool support. Arguably, all of the work on call-graphs can be considered part of this section, as the call-graphs themselves are not the piece of the toolbox developers are missing, but the tools that rely on call-graphs. The work focusing on polyglot development is more nuanced in terms of benefit, and is explained in more detail below.

Improving developer experience when working with more than one language has benefits for more than just developers who are already writing polyglot programs and contending with extra sources of overhead. It will also free all programmers to become polyglot developers, by reducing the pain in doing so. This will have important effects. Developers will be able to make better use of all the tools at their disposal. Rather than needing to search Google and ask StackOverflow about "NumPy for C++" or "Boost for Python," polyglot developers could use the libraries they're familiar with. This can lead to faster development time, since developers do not need to spend as much time becoming familiar with libraries that are similar-but-not-quite-identical to the ones they are accustomed to. Also, it may even lead to fewer mistakes in using the APIs, which can both speed up development time and create less buggy code. Developers using an almost-identical API can be tempted to treat it as completely identical, and subtle mistakes can creep into the code. If developers can use the API they are familiar with, instead of a similar one, these mistakes do not occur. In addition to the extra library support, more support for polyglot development encourages developers to switch between languages dependent on the task at hand. Doing so will allow developers to write more natural code for each task, and will hopefully speed up development time. Rather than being forced to weigh the relative trade-offs and benefits of possible development languages on the broad scale of a full project, developers can weigh these trade-offs at a more fine-grained level, such as the function level. This will allow them to reach a more optimal development flow, since they are never "locked in" to using a particular language because of legacy code, library support available, or the most helpful language for the majority of the project. In a world where polyglot programming is faster and easier, developers will be able to switch between languages as the task they are working on changes.

Another important benefit of polyglot development is collaboration. Developers developers work more effectively and efficiently in the language they are most comfortable in and prefer, but polyglot development also opens the door for collaborations between different areas. There are many areas of science with pet languages. Fortran is unusually ubiquitous among meteorologists, for example. Yet most of the programming community shies away from Fortran, meaning it does not have many of the most up-to-date libraries and techniques that the rest of programming world is using. Meteorologists are then effectively deprived of many

tools and techniques that should be at their disposal. On the other side, they also develop interesting frameworks and predictive models that other communities, such as the machine learning community, might benefit from . . . if they were accessible from a language other than Fortran. Meteorologists are just one example: many other disciplines have pet languages and are similarly isolated from programming communities and resources. Many scientific disciples have one or two favored languages that might isolate them from other scientific research. Bridging this gap by making polyglot development easier can better equip scientists for cross-disciplinary research, and allow them to share broader-impact ideas and techniques that are typically only accessible from one language. Collaboration and idea-sharing are pillars of modern research that more straightforward polyglot development would directly support.

1.4 Dissertation Structure

The remainder of this dissertation is organized as follows. Chapter 2 discusses our work on statically analyzing C programs to uncover developer intent about length information for arrays. Chapter 3 discusses our approach to constructing call-graphs in JavaScript. Both Chapter 2 and Chapter 3 contain a discussion of their related works and suggestions from the authors for future work. Chapter 4 summarizes the results from Chapter 2 and Chapter 3, and presents future directions for work more broadly than discussed in the individual chapters. We also provide advice for other program analysts facing similarly stymieing experimental evaluations, and conclude. Appendix A contains data from GitHub projects categorized by Two Sigma Ventures [47] as the current “most popular” GitHub repositories.

Part II

Polyglot Tools

2 C ARRAY LENGTH INFERENCE

Substantial portions of this chapter are derived a paper by Maas et al. [25] published in ASE '16.

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 2.4.3.2), 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, which have minimal type information. A C array is simply a raw pointer to allocated memory that may (or may not) 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.

The remainder of this chapter is organized as follows. Section 2.1 establishes the motivations for automating annotation inference, and describes the annotation system that consumes our analysis results. Section 2.2 reviews related work to set the context for our novel approach. In Section 2.3 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 2.4 assess the effectiveness of our implementation when applied to multiple real-world libraries. Section 2.5 discusses options for future work and Section 2.6 concludes.

2.1 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 identify C array arguments and recover their lengths from LLVM [20] 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.

```
1 def foo(array, string, fixedLen):
2   x = c_expectLenArg(array, len(array)
        )
3   nulSafeArray = string.replace('\0',
        '')
4   nulSafeArray.append('\0')
5   y = c_expectNulTerm(nulSafeArray)
6   z = c_expectFixedLen(fixedLen)
7   return (x, y, z)
```

Listing 2.1: Calls using language bindings from Python to C without length inference

```
1 def foo(array, string, fixedLen):
2   x = c_expectLenArg(array)
3   y = c_expectNulTerm(string)
4   z = c_expectFixedLen(fixedLen)
5   return (x, y, z)
```

Listing 2.2: Calls using language bindings from Python to C with length inference

Listings 2.1 and 2.2 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 2.1 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 2.2, the user directly passes the arrays and string without extracting length information. In Listing 2.1, 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 2.1 and 2.2 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 2.2. 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 [35] (see Section 2.2). 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. Furthermore, 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 2.2 than those in Listing 2.1. 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 [12]. 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 chapter 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.

2.2 Related Work

Ravitch et al. [36] automatically generate bindings based on static analysis of C, while Ravitch and Liblit [35] 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 [14] 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 [7] 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. [24] 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 [29] 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 relies on complex symbolic predicates to facilitate a precise approach. They require precision in order to accept only safe memory accesses. Dhurjati et al. [5]'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 [21] 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 [27] 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. Furthermore, 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 [38] use an interprocedural bounds analysis to determine memory safety, and their technique has a wide variety of applications. Their approach is similar to that of the symbolic range analysis performed by Nazaré et al. [28], 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 useful for our algorithm, we take a static approach, and thus cannot make use of dynamic information.

SWIG [3] 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 does it handle the lengths of arrays. Thus, we target GObject-Introspection instead.

2.3 Approach

This section is organized as follows. Section 2.3.1 describes the formal definitions of each of our length properties, and Section 2.3.2 introduces the assumptions we leverage to approximate these length properties. Sections 2.3.3 to 2.3.5 describe the analyses used to recover length information, and Section 2.3.6 describes our method for merging length properties when more than one strategy appears to be used for encoding the length. Section 2.3.7 discusses expanding our analysis to structure fields. Finally, Section 2.3.8 explains sources of false positives and true negatives in our analyses.

2.3.1 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.

2.3.2 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 2.3.1) 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 2.3.1. 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. Furthermore, 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.

2.3.3 Symbolic Range Analysis

Our techniques (especially those in Section 2.3.4) 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. [28], handles only integer values and allows interval expressions to be symbolic.

For a practical explanation, consider Listing 2.3, 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 the 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

```
1  int symbolicLength(int *array, int y)
2  {
3    if (array[3] < 0)
4      return array[0];
5
6    int sum = 0;
7    for (int i = 0; i < y; i++)
8      sum += array[i];
9
10   return sum;
11 }
```

Listing 2.3: Example function with an argument of symbolic length

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 2.3.4.

2.3.4 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 2.3.1. We are attempting to identify exactly the length in case 2. Consider the function in Listing 2.3. 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`. 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 2.3.1. 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 2.3, 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*) [28] 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 2.4, 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

```
1  int symbolicLoop(int *array, int y)
2  {
3    int *end = array + y;
4    int sum = 0;
5    while (array < end) {
6      sum += *array;
7      array++;
8    }
9    return sum;
10 }
```

Listing 2.4: Example function with an argument of symbolic length containing a specialized loop

definition of symbolic-length from Section 2.3.1. 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, rather than symbolic. If a symbolic 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: Items 1 and 2 in Section 2.3.1.

2.3.5 Sentinel-Terminated Detection

Per Section 2.3.1, 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 2.5 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 2.5, 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`.

```
1  guint g_str_hash (gconstpointer v)
2  {
3    const signed char *p;
4    guint32 h = 5381;
5
6    for (p = v; *p != '\0'; p++)
7      h = (h << 5) + h + *p;
8
9    return h;
10 }
```

Listing 2.5: Real-world function with an argument sentinel-terminated by `NUL`, taken from `glib`

Let the entry of the loop be $L_{\texttt{entry}}$. Let $check(\texttt{array}, i, \omega, b)$ be an access of `array` at offset i, comparing the value at this offset to ω with Boolean result b. We consider $check(\texttt{array}, i, \omega, b)$ 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 $L_{\texttt{entry}}$ looping back to $L_{\texttt{entry}}$ 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.

Notice a deviation from our formal definition of a sentinel-terminated array in Section 2.3.1. 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.

2.3.6 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 2.3.1 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

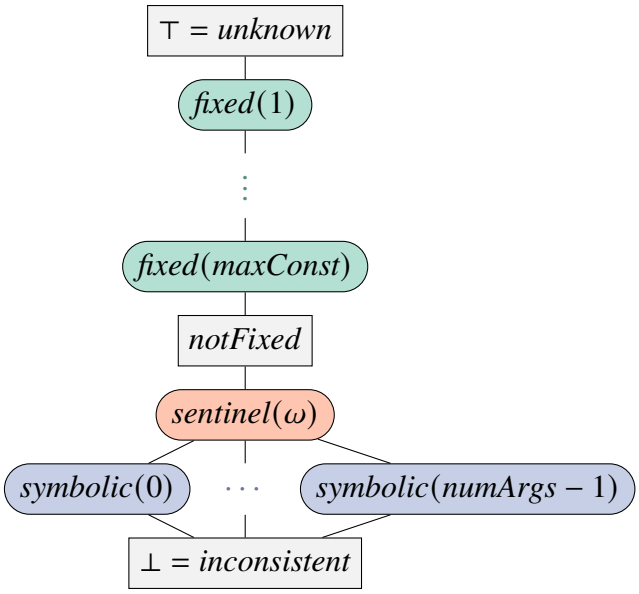

Figure 2.1: Result lattice for any single array under analysis in a function with *numArgs* arguments and no constant index larger than *maxConst*. Ellipses notwithstanding, the lattice is finite in both width and height.

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 2.3.2), 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 2.1 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 $fixed(n)$ denote a fixed-length type where `n` is the fixed length of the array. Similarly, $symbolic(n)$ 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 of the form $\textit{fixed}(n)$ 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 2.1. 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.

2.3.7 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 2.3), 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 2.4.5. This overhead is likely because of the vast number of stores into structure elements in a sizable library. Furthermore, 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.

2.3.8 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 reporting an erroneous length type; completeness requires reporting all length types. We sacrifice both soundness and completeness in favor of practical utility.

2.3.8.1 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 2.3.6 that each array has a most general annotation. Consider a *false positive* to be a function argument annotation identifying an "incorrect" length: i.e., any annotation but the correct most general 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 $\textit{fixed}(8)$, reporting length $\textit{fixed}(7)$ 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 length properties that are less general than the correct one, but this incurs a false positive in addition to a false negative.

Table 2.1: Library details. KLoC measures thousands of lines of source code, estimated with SLOCCount [50].

			Number of Function Arguments				
Name	KLoC	#Funcs	All	Symbolic	Fixed	Sentinel	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

2.3.8.2 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 $\texttt{y} - 1$ elements are accessed from `array`, and the $\texttt{y}^{\text{th}}$ 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

Table 2.2: 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

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.

2.4 Experimental Evaluation

We have implemented the analyses described in Section 2.3 using LLVM 3.7 [20]. 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.

2.4.1 Test Subject Selection

We have evaluated our tool on the following libraries:

- `gck` v3.18 implements PKCS #11, a form of public key cryptography [10].
- `gio` v2.46.2 is a virtual file systems API [11].

- `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 [45].
- `libssh2` v1.6.1 is an implementation of the SSH2 protocol in an extensible C framework [13].
- `telepathy-glib` v0.23.3 is a D-Bus framework for real-time communication [46].

Most of these libraries are part of the GNOME Project [9] 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 2.1 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 even 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: instead, the number of arguments listed under the "all" column indicate the total number of arguments to each function. Those that are not assigned to symbolic, fixed, or sentinel either are not array types, or follow some other length convention (variadic type, inconsistent use of our length types, etc). 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 present the true positive rate and false positive rate of each type of length information in Tables 2.2 and 2.3. 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

Table 2.3: 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

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 2.3.8.2.

2.4.2 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 annotate manually. Table 2.2 summarizes our findings. Over each of the full libraries, our automatic analysis achieves 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 2.4.4. 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 2.3.4 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

Table 2.4: 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

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.

2.4.3 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.

2.4.3.1 Automated Analysis

Table 2.3 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 2.2. 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 2.3.8.2, 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 2.3). 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.

2.4.3.2 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. To conduct this analysis, we manually determined the correct annotation when the human-provided annotations differed from the ones our tool provided. This often required tracking the array arguments as they were passed through numerous function calls. We blinded ourselves to which answer was emitted by which source, and in some cases determined that both answers were wrong. While it is possible that our tool and the human annotators were incorrect in the same way on some arguments, given that they examine the code very differently, it seems quite unlikely that this would occur commonly.

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 2.4 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,

```
1  static inline gboolean
2  contains_non_ascii (const gchar *str, gint len)
3  {
4    const gchar *p;
5
6    for (p = str; len == -1 ? *p : p < str + len; p++) {
7      if ((guchar)*p > 0x80)
8        return TRUE;
9    }
10   return FALSE;
11 }
```

Listing 2.6: Real-world function with inconsistent treatment of an argument's length, taken from `glib`

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 submitted bug reports for the remainder of the human errors our analysis detected in the library `gio`, which was addressed by the developers.[1][2]

2.4.4 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 2.6, `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 `NUL`s. 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 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 2.1 in order to combat this issue. By combining our sentinel-terminated and symbolic-length analyses, we produce only the more general symbolic-length annotation. When our analysis fails to detect that the

[1] `https://bugzilla.gnome.org/show_bug.cgi?id=765063`
[2] `https://bugzilla.gnome.org/show_bug.cgi?id=787812`

Table 2.5: 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

array may have a symbolic length under some conditions, we may produce false positives (of sentinel-terminated) for these arrays.

2.4.5 Structure Information Results

We extended our implementation to analyze structure fields as described in Section 2.3.7. We only found different results in the sentinel-terminated case, so we just report these results. Table 2.5 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.

Furthermore, 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 2.4.1 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 2.5) suggest that the marginal benefits may not make structure analysis worthwhile.

2.5 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 client code

might be the library itself, which often calls 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 [14].

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 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 2.6 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.

2.6 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.

Part III

Dynamic Typing Tools

3 JAVASCRIPT CALL-GRAPH CONSTRUCTION

This work was done in collaboration between Alisa Maas, Julian Dolby, and Ben Liblit.

3.1 Motivation

As JavaScript becomes increasingly popular, confidence in JavaScript programs becomes critical. JavaScript developers need to be able to effectively debug, understand, and prove safety properties about these programs just as much as developers who work in other languages. Yet static analysis of JavaScript code is difficult and unscalable. Many of these analyses rely heavily on call-graphs for information regarding interprocedural edges. JavaScript's extremely dynamic nature confounds traditional static analysis approaches to call-graph construction, however. In a statically typed language, type information provides key insights that limit the number of possible edges at any given call-site. In JavaScript, types are defined primarily dynamically.

Traditional approaches to call-graph construction rely heavily on static type information that JavaScript developers are not equipped to provide. In lieu of static type information, call-graph construction in JavaScript relies heavily on a points-to analysis for each object containing a function that might be called. These points-to sets represent an over-approximation of each object's type. Any changes to these sets have cascading changes in the call-graph, meaning that as the points-to set for each callable object grows, so does the work required to generate the call-graph. Thus, each time imprecision is introduced into one of these points-to sets, the call-graph not only becomes less precise, but also requires more work to generate. Most state-of-the-art approaches deal with this issue by improving the points-to sets. Often they exploit domain knowledge or introduce controlled sources of unsoundness or incompleteness. Our approach instead focuses on identifying sources of ambiguous points-to sets and the critical range in the code in which this effect could cascade into a state explosion. Equipped with this knowledge, we are then able to avoid the state explosion by reanalyzing the critical range once for each value in the points-to set. By using this approach, our methodology performs a little extra work up-front to avoid doing much more work later.

The remainder of this chapter is organized as follows: Section 3.2 discusses current analysis techniques that we will build off of to create our approach and Section 3.3 discusses related work. We discuss our approach in Section 3.4, and outline our implementation and

```
1  function foo() {}
2  function bar() {}
3  function extend(a, b) {
4      for (var x in a) {
5          b[x] = a[x];
6      }
7  }
8  var object = {};
9  object.function1 = foo;
10 object.function2 = bar;
11 var clone = {};
12
13 extend(object, clone);
14 clone.function1();
```

Listing 3.1: Common JavaScript function where precise points-to sets are not sufficient for a context-insensitive analysis to compute a precise graph

evaluation in Section 3.5. In Section 3.6 we discuss future directions for research, and we conclude in Section 3.8.

3.2 Background

The vast number of possible paths through JavaScript programs means that a full, context-sensitive, flow-sensitive approach generally does not scale. As a result, most static JavaScript or points-to analysis tools are partially context-sensitive and flow-insensitive. We discuss some of these tools further in Section 3.3. We will be focusing on WALA's analysis engine in particular, as we implemented context-splitting as an extension to WALA. However, we expect that our techniques (and results) will generalize to similar systems.

In WALA's intermediate representation (IR) of the program, nodes are (Context, Function) pairs and are organized into a contextual control-flow graph (CCFG): a control-flow graph augmented with contextual information. Contexts are containers for any state information that might add helpful, but limited, context sensitivity. Each node is analyzed just once, but as the amount of context sensitivity applied to the analysis grows, the number of CCFG nodes increases as well.

To understand the motivation for why limited context sensitivity may be needed, see Listing 3.1. Perhaps surprisingly, using contexts at a function-level does not allow most analysis tools to determine that the value for `x` is the same on both sides of the assignment in line 5. Without contextual information, most analyses will assume that *all* of `b`'s fields may alias *any* of `a`'s fields, rather than only the corresponding fields. In this example, while a human analyst could easily determine that the function call in line 14 could only ever call `foo` (since it is copied from `object`'s `function1` field), a programmatic approach is likely to conclude that *either* `foo` or `bar` is called, without being able to distinguish between the two.

While in this case, only a small amount of imprecision appears to be incurred here, actually quite a lot of imprecision can arise from loops like this one, as it is a common pattern in JavaScript for cloning objects, subclassing, or even instantiating new objects of a conceptual class. The fields being copied tend to represent instance methods. So even a seemingly-innocuous loop such as this one can be responsible for substantial imprecision in the call-graph. At worst, this leads to unacceptably high run-times. Even at best, it still leads to far less useful call-graphs, as it loses crucial information about possible call-graph edges.

Prior work [44] partially addresses this source of imprecision using 'correlation tracking'. However, Sridharan et al. [44]'s approach accomplishes this by extracting `for-in` loop bodies into their own methods. They then rely on argument sensitivity to achieve the desired precision. While this does allow the analysis to handle this case, it has a few drawbacks. It introduces an extra function body per `for-in` loop, introducing new complexity into the call-graph, and potentially making it difficult to recover the original line numbers of code near the `for-in` loop body.

This approach also does nothing to handle other sources of imprecision in a similar situation. For example, imagine a similar assignment to the one in line 5, where the field access has several possible values, yet is the same on both sides of the assignment. Even though the only difference would be the presence or absence of a loop body, the correlation-tracking approach would need substantial modifications to identify this new pattern, and would introduce many, many, new interprocedural edges in the process, complicating analysis and making the resulting call-graph less useful for the end user. This approach inspired us to consider whether we could generalize something similar to, but more flexible than, correlation tracking *without* modifying the structure of the code.

3.3 Related Work

Call-graph construction for JavaScript is not a new problem, but neither is it a solved problem. The complicated challenge of state explosion inspires many different approaches. The difficulty of Andersen-style analysis for JavaScript has motivated work in approximate algorithms, in which accuracy and completeness are traded for scalability. One approach is a *field sensitive* analysis, in which individual properties are modeled with a single abstract location. This can greatly reduce the amount of propagation needed, especially when combined with deliberate unsoundness to avoid expensive cases [6].

Andreasen and Møller [2] exploit static determinacy information to help with the state-explosion issue. This technique simplifies code whose execution always results in the same value, and could be applied alongside our approach to improve precision further. A similar idea inspired dynamic determinacy analysis [41], which used a combination of lightweight static analysis and execution traces to find deterministic values, which were applied to improve analysis. Ko et al. [18] tune their static analysis to a particular set of executions (presumed to generalize to other important executions); our use case is somewhat different, in that we are providing fully static call-graphs that always generalize to all executions.

Sridharan et al. [44] focus on improved analysis for `for-in` loops in JavaScript, by using loop-sensitivity to distinguish iterations of such loops from one another. This analysis inspired our approach, but the implementation of their work limited its applicability, as discussed in Section 3.2. Park and Ryu [33] likewise focus on distinguishing loop iterations from one another with the aim of reducing imprecision. Our work builds upon this idea to create a generalized context-splitting framework (see Section 3.4.2 for more details). Park et al. [34] use regular expressions to better bound the possible values for strings, which can cause the state space to explode when they are used as keys to access property fields of objects in JavaScript. This approach is complementary to our own; it focuses on improving the points-to sets (which we rely on), but is not sufficient on its own to handle large-scale projects where imprecision creeps into the program through other sources.

Other tools focus on improving precision of different, but related, analyses in JavaScript. Jensen et al. [15] recover type information for JavaScript. This can partially address the state explosion by reducing the number of options at callsites, though this may not be effective when the types are convoluted or difficult to discover. Jensen et al. [16] use a static analysis to determine the value of strings passed to `eval`, then refactor the code to remove them, allowing static analyses to soundly analyze even programs that originally contained `eval` statements. Our implementation is in WALA which, like most JavaScript call-graph-construction tools, assumes that `eval` statements are no-ops. By replacing eval statements with straight-line code, this work could serve to make our call-graphs more sound and more complete, but it does not directly address the issue of scalability.

Some tools focus on points-to analyses. Kastrinis and Smaragdakis [17] mix different sorts of context-sensitivity to produce better analysis results. Their approach works by first performing a context-insensitive analysis, using this to identify sources of ambiguity, and adding selective context sensitivity where there are sources of ambiguity. Our work also performs selective context sensitivity, but we focus on limiting the range of the sensitivity

and craft policies to determine sources of ambiguity, and we do not attempt to perform a context-insensitive analysis to inform our approach. In JavaScript, a fully context-insensitive analysis of many real-world programs would be infeasible, since the imprecision leads to drastic runtime costs. Other researchers (such as Smaragdakis et al. [43], Wei et al. [49]) use static analysis to attempt to identify sources of state (and therefore runtime) explosion. Like Kastrinis and Smaragdakis [17], these tools focus on the points-to analysis component, and rely on the assumption that it is possible to create a less-precise analysis that terminates. Unfortunately, this is rarely true for JavaScript call-graph construction. Both tools use their analyses to offer personalized recommendations for call-graph-construction techniques based on likely sources of imprecision (as determined by their analysis), which would be very useful in combination with our approach.

Smaragdakis et al. [42] propose a set-based pre-analysis for points-to analysis. Liu et al. [23] also work in WALA and improved upon its points-to analysis by using a parallel and incremental fixed-point iteration. As their work was merged into the WALA source code, we are able to directly reap the benefits of the improved points-to analysis. Points-to set construction techniques are complementary to our work, as improved points-to analysis can improve the precision of call-graphs.

On the other hand, some existing analysis could substantially benefit from an improved and more flexible call-graph construction algorithm. For example, Wei and Ryder [48] construct a blended taint analysis built on top of WALA. For programs where the imprecision hinders the analysis, they may find our techniques beneficial in reducing the number of false positives, or simply increasing the number of programs they can analyze. Likewise, MinerRay [37] attempts to identify WebAssembly and JavaScript programs that hide cryptojacking attacks, and builds a call-graph in order to do so. A more precise and scalable call-graph construction technique could further boost their results and reduce potential false positives.

In other domains, the Astrée static analyzer uses trace partitioning [26] to more precisely analyze C programs by partitioning the abstract domain into separate traces, typically distinguished by control-flow information. Their emphasis is on numerical precision, particularly on ranges of values, and operates over the domain of C programs, while ours is designed for JavaScript pointer analyses. Furthermore, our approach need not maintain control-flow path information, when we do not consider it relevant to the task of pointer analysis. (It also has the flexibility to maintain select control-flow information as it deems relevant.) The concept is similar, but because they target a different problem, their strategies for maintaining partitions (and especially for merging them) differ from ours.

Inspired by trace partitioning, recent work by Nielsen and Møller [30] focuses on a special case of it, value partitioning. Value partitioning, like context splitting, is designed to target JavaScript analysis, but while we choose to analyze select ranges with partitioned values, they choose to partition the abstract values themselves. Both approaches have strengths and weaknesses, but one strength of analyzing the code separately per each abstract value is the possibility of maintaining control-flow information should it prove helpful. Value partitioning considers the effect of partitioning based on type, where we do not assume any type information is present; likewise, we provide data about partitioning (splitting) on phi nodes, which the current implementation of value partitioning does not address.

Our approach is designed to be sufficiently flexible that new automated strategies for determining partitions/splits can be performed without modifying the underlying analysis engine at all, simply by implementing interfaces, and did not require substantial modification to our underlying analysis engine. In fact, the only changes we made to WALA's algorithm was to change the visibility on some methods from private to public or protected, and to mirror methods with new options where we add flexibility. In contrast, the value partitioning approach requires modifying the underlying static analysis in order to prepare for adding value partitioning to a new engine, and again each time a new strategy is constructed. This may prove daunting to researchers seeking to test out their own strategies, as it would require them to understand the underlying engine well enough to modify it, and also will cause them to diverge from the official engine. In effect, our approach strikes a balance between trace partitioning and value partitioning: we allow, but do not require, contextual information to be carried along with more precise information about values of interest. We provide a flexible approach without requiring heavy modifications to a traditional static analysis for JavaScript in order to further extend our tool.

3.4 Approach

Our ultimate aim was to introduce new, more flexible configurations for context sensitivity into WALA to test whether contextual information at the level of an abstract value provides any real-world benefit. The implementation itself required minimal modification to WALA.

3.4.1 Context Splitting

At its core, context splitting is made up of two key components: context splitters, and split range policies. Conceptually, a context splitter determines which values (which we called

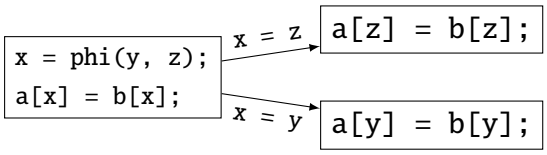

Figure 3.1: Context splitting in action. The edges represent the dataflow and logical connection between the split contexts, and the code boxes represent the contexts created by context splitting.

split points) may be a source of ambiguity, and a split range range policy determines the critical window of opportunity when this ambiguity may be propagated (which we call the *split range*). Split ranges are defined with respect to a particular split point; the code in the split range is analyzed once for each possible value that the split point may take on.

Formally, we can define a *context splitter* as a function that accepts a representation of a JavaScript function and returns a set of *split points*: abstract object definitions in the given function. Likewise, a *split range policy* is a function that accepts a representation of a JavaScript function and a split point, and produces a *split range*: a subset of instructions from the function. These instructions must be dominated by the split point, but they need not be *every* instruction dominated by the split point. Note that the instructions contained in a split range need not be contiguous; they must be dominated by the split point and contained in the same function as the split point, but other than that, we place no restriction on the elements in the split range. This flexibility allows us to include only the instructions that may propagate ambiguity and ignore nearby, unrelated instructions, cutting down on the amount of extra analysis we require.

Prior work tends to analyze each function once per context, which means that each context represents key state information that requires separate analysis of the entire function. However, for efficiency, we wish to allow our additional contexts to only hold sway over part of the function; specifically, the split range described by the split range policy. To split a function-based context into one with an arbitrary range, we need to do a few things. First, we must create special contexts (*split contexts*) for each possible value of each split point. Next, we must partition responsibility for analyzing the function among these split contexts. Finally, we must fix up the dataflow between the partitions of the analysis. We will discuss the creation of split contexts and their use in Section 3.4.1.1, and the dataflow in Section 3.4.1.2.

To understand the benefit that could be gained from context splitting, consider Figure 3.1. This demonstrates a very simple instance of context splitting, where the split point is the definition of `x`, and the split range is simply the second line. The `phi()` indicates an assignment

that would be a phi node edge in the IR, which often proves a source of ambiguity. It represents a definition where the value of `x` can be either the value for `y` or the value for `z`. As with the `for-in` loop, an approach with only function-level context sensitivity cannot identify that the value of the abstract object for `x` must be the same on both halves of the assignment in the second line. However, context splitting produces two contexts. Each analyze the second statement with contextual knowledge of the value of `x`. Thus, when pooling their combined information, it becomes clear that there is no situation where the fields from `b[y]` could be copied to `a[z]`, so this reduces the state space compared to an algorithm lacking this contextual information.

```
1  function returnY(x, y) {
2      return y;
3  }
4  function safe(x, y) {
5      return 0;
6  }
7  function unsafe(x) {
8      //TODO: make sure doSecret
9      //never gets here!
10     x();
11 }
12 function doSecret() {
13     //...NOTE: do not call
14     //this from unsafe()!
15 }
16 function doUnsecret() {
17     //...code body
18 }
19
20 function main() {
21     var check;
22     var i = Math.random(0,1);
23     var x = doUnsecret;
24     var y = doSecret;
25     if (i > 0.5) {
26         check = returnY;
27         y = doUnsecret;
28     }
29     else {
30         check = safe;
31     }
32     var realCheck = check;
33     var realY = y;
34     unsafe(realCheck(x, realY));
35 }
36
37 main();
```

Listing 3.2: Example for demonstrating context splitting

3.4.1.1 IR Control

Analysis is usually done at the intermediate representation (IR) level; we will be assuming that the IR is a directed graph of instructions in SSA form. Unlike past approaches, which analyze a function's IR once for every context it has, we wish to delegate the responsibility of deciding what to reanalyze to the split range policy. To do this for a given function, we create one partition of that function's IR for each split point, plus one "root" partition for all parts of the IR that fall outside of all split points. We organize the split points into a forest of trees

based on the control-flow dominators graph: that is, a "parent" split point must execute before any of its "child" split points may. Each node in one of these trees is responsible for analyzing some portion of the IR.

Let `split` be a node in the split point forest, `IR.nodes` be the set of instructions in the function under analysis, `split_range` be the split range policy, and `descendants(x)` be a function returning `x`'s descendants in the split point forest. Then we can define the function determining which portion of the IR that `split` controls as follows:

$$\texttt{IR(split)} = \{X \in \texttt{IR.nodes} \mid X \in \texttt{split\_range(split)} \wedge$$

$$\forall \texttt{child} \in \texttt{descendants(split)}, X \notin \texttt{split\_range(child)}\}$$

In the case of two different split ranges that contain the same instruction, the split point closest to the instruction in the dominators tree gets control of the IR. Any remaining parts of the IR not controlled by a split site tree is covered by a single, top-level context.

Because the contexts are nested based on control-flow dominance, our context-splitting is "multiplicative" in the sense that each possible permutation of values from all split points are explored. If two split points have an overlapping range, each time the overlapping range is analyzed, the analysis is performed using an abstract value from each points-to set. Therefore, if one split point has n possible values, and another has m, any instructions in both split ranges is analyzed $n * m$ times. The complexity our technique adds depends on the number of split points, how many abstract values are in each of their points-to sets, and how often their ranges overlap.

We assume that split points are all abstract object definitions, though the only restriction we place on the abstract objects is that their representation must have finitely many possible values. For example, WALA's abstract objects have only one value for integers; however, our strategy would still operate the same if we instead used a call-graph construction framework that has separate abstract values for 0 and non-zero integer values, or a framework that has separate abstract values for each possible 32- or 64-bit integer value. As we discover new abstract values for each split point, we create split contexts for each one. Note that in any correct, terminating, points-to analysis, there must already be a finite, though unbounded, number of abstract values for any abstract object definition, so we will only create a finite number of split contexts. Since we analyze each split range once per split context, the number of abstract values in the points-to set of each split point bounds the extra analysis passes we introduce. In many cases, with the correct split range policy and context splitter, our

extra analysis passes can serve to *reduce* the total amount of analysis time by preventing the points-to set and call-graph from growing unmanageably large.

3.4.1.2 Dataflow

In order to properly finish performing context splitting, we must correct the dataflow through the new contextual control flow graph with the split contexts. For each split context, we must:

- Add a dataflow property defining the split point as the specific value this split context represents. Although this property is not globally true, we will create a split context for each possible value for the split point. Therefore, as long as we properly merge the analysis results, we do not introduce additional unsoundness or incompleteness.
- Add dataflow edges from outside the split range for this split context to inside. Edges must be added wherever a value defined outside the split range is used within the split range. This insures that if there are updates during analysis, they are propagated to the split context; otherwise, we will miss important opportunities to make our analysis more precise.
- Add dataflow edges from inside the split range for this context to the IR outside. Edges must be added whenever a value inside the split range is used outside the split range. Since we expect to have multiple abstract values for each split point (otherwise there is no point to splitting), we must merge their analysis results at this point. The safest option is to merge the analysis results by taking the union over all results, as adding results not in this union would be unsound. It may be possible to prune out some results that are infeasible, but our technique is not designed to look for infeasible paths, and so we ignore them, and merge the analysis results by taking the union over all results.

3.4.1.3 Toy Example

To see where context splitting would reduce ambiguity, consider Listing 3.2. Note that lines 32 and 33 are primarily there to illustrate the phi nodes that are created at roughly this point in the IR: a developer would likely not write these statements directly, but in the code as it is being interpreted by the context selector, they will exist. Suppose that `doSecret()` interfaces with sensitive data, as the name suggests, and `doUnsecret()` does not. If we notice the TODO in line 8 and want to ensure that `doSecret()` never makes its way to `unsafe()`, we need quite a bit of contextual information not provided by context sensitivity at the function level. `unsafe()`

is only called in line 34, so it really matters whether the value returned from `realCheck` may ever be `doSecret()`. The two possible targets for `realCheck` are `safe()` and `returnY()`. As the name suggests, `safe()` does not expose `doSecret()`, but `returnY()` does when `y` is `doSecret()`. As you can see, the only case in which `check` could be `returnY()` is if we have ensured that `y` is set to `doUnsecret()`. Yet, lacking contextual information, WALA (as well as many other call-graph construction techniques), will not recognize that `y` is always set to `doUnsecret()` whenever `returnY()` may be called.

Context splitting can help cut through the ambiguity. In line 32, the value for `check` would make an excellent choice for a split site. There's a relatively small number of possible abstract objects (in this case, just two: either `returnY` or `safe`, depending on which branch was taken), and the range where precision is needed is quite small. All we really need the range to include is the instructions in line 34, which most likely will only amount to one basic block. Furthermore, if we add one more split site for the value of `realY` with the same range, we can use the multiplicative effect of nested split sites to determine that there is no path where `check` is `returnY` *and* `y` is `doSecret`. Although we do no pruning of infeasible paths, because of the dataflow for each context splitter being stitched together, these two split points are sufficient to eliminate any paths where `y` is `doSecret` and `check` is `returnY`.

This example seems perhaps rather trivial, but real-world code (though more convoluted) often contains similar sources of ambiguity, and in practice, developers rely on tools to report when their code may pass sensitive data to insecure functions. Determining whether this occurs can rely on contextual information at a finer granularity than the method as a whole (as in Listing 3.2).

3.4.2 Context Splitters

Each context splitter is responsible for determining a set of split points, which are value-definition points. Due to WALA's implementation, we require that split sites all occur at the top of a basic block or in phi nodes (which are maintained separately, but logically belong at the top of a basic block). WALA does support editing the CFG during analysis, so if it is critical that a node not at the top of a basic block be used as a split point, it is possible to split the existing basic block into two, such that the desired node is at the top. None of the context splitters below modify the CFG. If any nodes match the policies below, but are in the middle of a basic block, they are ignored. Control flow will be analyzed separately for each possible abstract value for the split point, so it is important that these split points represent nexuses of imprecision. The context splitters we implemented are as follows:

- **Correlation Tracking Splitter.** The correlation tracking splitter was modeled after WALA's existing technique of the same name [44]. It selects all `for-in` loop variable definitions as split points. Prior work on WALA implemented a solution to this imprecision, [44] but it relied on modification of the call-graph, pulling the loop body into a new function and relying on its interprocedural analysis. This splitter addresses the same core problem, but without a need for making the CFG more complex by introducing a new function. With context splitting, the same goal can be achieved more reliably and without modifying the call-graph (which could affect other analyses), simply by identifying `for-in` loop variables as requiring context sensitivity.

- **Field Read Splitter.** As the name suggests, the field read splitter selects all definitions whose RHS is a field-read operation. Depending on the code being analyzed, this may end up creating many split points. Yet because field reads are commonly sources of imprecision, the benefit provided by context sensitivity at these program points might outweigh its costs. The simple reason for this is that often in JavaScript code, the value from a field read is used to copy a field from one object to another. As with correlation tracking, stock WALA's analysis is not context-sensitive enough to recognize that the field access on the left-hand and right-hand sides will resolve to the same value. Therefore, adding additional context sensitivity to these values could be of potential interest. Of course, this selector comes with the drawback that it also may incur additional cost, as there may be many field reads in the code that do not truly require context sensitivity.

- **Phi Node Splitter.** Like the field read splitter, the phi node splitter is very general. The set of split points connected by this policy is simply the set of all phi node definitions. Phi nodes often represent sources of imprecision, though due to the sheer quantity of them, the phi node splitter runs the risk of creating more extra work than it can resolve.

- **Special Name Splitter** This splitter is primarily intended for users who have specific value definitions that they know cause imprecision in the call-graph. It examines the name of each variable, and creates a split site for variables whose names end in "$split." We do not present results for this splitter, as it is intended as a diagnostic tool more than as a general-purpose addition to CFG-construction algorithms.

Each of these splitters are designed to get at particular sources of imprecision, but of course each is only one piece of the puzzle. Now that we have identified *which* extra contextual information we need, we must determine *where* we need it.

3.4.3 Range Policies

Split range policies are responsible for determining, given a split site, how long its context sensitivity should last. Currently, each context splitter must designate its own split range policy rather than allowing the policies to vary situationally, though this is something we may revisit in future work (see Section 3.6). A split range policy can be thought of as a map from split point to a set of instructions for which that split point requires context sensitivity. The three policies we have implemented are described below, in order from "largest" to "smallest."

- **Dominator Range.** For a given split point, the dominator range is all of its children (transitive or otherwise) in the dominator tree. Effectively, this policy is the most broad, and will contain the greatest number of instructions compared to the other two policies. This range contains no "holes," but may contain instructions that have nothing to do with the split point, possibly causing the analysis to perform extra work. On the other hand, due to its size, it is much less likely to miss a piece of the function where the extra contextual information might have been helpful. We recommend using the dominator range only for context splitters that are deemed crucial and for which missing contextual information could lead to a dramatic increase in analysis time. We also recommend limiting use to a small number of split sites.

- **Transitive Must-Use Range.** The transitive must-use range is more restrictive than the dominator range. Given a split point, its transitive must-use range is all instructions in the call-graph that (directly or indirectly) use the value in the split point. Unlike the dominator range, the transitive must-use range need not be a contiguous set of statements; there may be gaps where there are instructions that are unrelated to the split point. It could miss some places where contextual information might have been useful, but the most important locations are typically covered. We recommend using the transitive must-use range for most context splitters, as it provides a good balance between additional work required and providing contextual information where needed.

- **Must-Use Range.** The must-use range is very similar to the transitive must-use range. As the name suggests, the primary distinction is that the must-use range contains only direct uses of a split point rather than also including anything that transitively uses one. This does make it more likely to miss some places that contextual information would have been ideal, but as the smallest of the ranges, it also reduces the amount of additional work needed to perform the analysis. The must-use range should be used for situations where context sensitivity is needed only fleetingly.

3.4.4 Custom Policies

Given the specific needs of a program, a developer may decide that none of the existing context splitters and range policies properly identify the context sensitivity needed. Likewise, additional research may reveal new context splitters and range policies that outperform our existing ones. To take that into account, we provide a Java API to easily construct custom policies. Context splitters and split range policies can be constructed independently, so the developer can use a pre-existing splitter with a new range policy or a new splitter with a pre-existing range policy.

To construct a custom context splitter, we provide a superclass for developers to subclass, with one method that they must override. It provides the same information that WALA uses to select contexts at a method-level granularity, with an abstract representation of the following: (1) The method to provide contextual information for, (2) the actual parameters to the method, (3) the calling context (method + callsite). Given that information, the method expects a context to be returned. We provide a handy method which expects a set of instructions, representing split sites. All the developer needs to do is decide which program points are possible sources of ambiguity and package them up in a set. This can be decided by any features of the method they desire, or even based on the calling context.

As for context splitters, we provide a superclass for easy creation of custom split range policies. Split range policies are required to implement a method that takes as input a split site and returns a set of basic blocks (the split range for that split site).

Using a custom policy, a developer can have direct control over the amount of context sensitivity provided by the analysis at a far finer granularity than previously allowed. This customization allows developers to tailor the context sensitivity used to the specific needs of their programs. This means they can leverage known sources of imprecision to put context sensitivity precisely where needed, without overwhelming the analysis.

3.4.5 Combining Policies

When it comes to choosing a context selector, sometimes more than one context selector could be helpful. Of course, the developer could write a custom context selector that effectively combines the logic of two or more context selectors, but this is cumbersome and leads to duplicated work. Thus, we allow the user to specify multiple context selectors when running the analysis. We require them to specify a split range policy for each, and each context selector can be chosen no more than once. The split sites used when combining policies are

the union of all split sites. If multiple context splitters select the same split sites, the only difference between the two is the associated range policy. We default to whichever range policy is specified with the context selected first. This allows developers to write simple context splitters and range policies, specific to particular needs, rather than writing a large and convoluted policy that may prove far more difficult to test in practice.

3.5 Evaluation

3.5.1 Effective Context-Sensitive Metrics

Comparing call-graphs sounds like a straightforward task. However, in the absence of ground truth, and with call-graphs that are both under- and over-approximations, this task becomes more difficult than it initially appears. Thus, to show the correctness of our approach, we demonstrate that it does not make a call-graph construction algorithm less correct, and additionally compare the number of completed call-graphs on real-world programs between different policies and with the standard WALA analysis techniques. Next, we evaluate our work on a series of 49 popular, real-world websites and compare with WALA without our technique to demonstrate the empirical benefit gained.

3.5.1.1 Small-Scale Experiments

To ensure that the call-graphs created by our context splitting do not affect correctness of the call-graph, we compared the call-graphs our modified version of WALA produced with those the unmodified version of WALA produced. We chose 7 real-world websites primarily for 3 important features.

- We wanted to make sure the websites were human-readable. Many modern websites obfuscate their JavaScript code in an attempt to make their websites load more quickly, and/or to improve security. Of course, the developers who would need the call-graphs, those actually writing the code, are not operating on such deliberately mangled code. As it is important to assess whether any edges that context splitting adds or removes are accurate, we wanted code that we could manually inspect to verify correctness.
- We also ensured that the amount of JavaScript present in the chosen websites was non-trivial. If the only JavaScript in a given webpage was a few simple calls to

framework methods, it does not demonstrate anything if both call-graphs are identical, given that they would be nearly empty.

- Finally, we wanted to make sure that the websites were chosen *after* we finished our implementation of context splitting, to reduce the possibility of overfitting to those specific websites.

To that end, we chose 7 websites. In order to attempt to find websites matching these criteria, we looked through numerous lists of old websites that are still in operation, as many such old websites did not obfuscate code. For each that we found, we examined the source code manually to ensure that a sufficient amount of JavaScript code was present, and that it was human-readable. We also did not conduct this search, or analyze these websites, until after our context-splitting approach was fully implemented. While it is theoretically possible that the use of older JavaScript code may not capture errors our approach will only cause on more modern JavaScript features, we attempted to mitigate the likeliness of this using unit tests to cover more modern features. In every case, WALA with and without context splitting produced identical call-graphs. Given that the scale of these websites was much smaller than the typical modern project, this is not a surprising result, nor does it disparage any help context splitting could provide in other websites. We intentionally sought out simple enough websites that WALA was able to finish and print out a call-graph in under a minute. We also chose a website from our list at random, and manually verified about a quarter of the control-flow edges the analyses found (chosen at random). All edges we examined were valid.

To ensure that our context splitting even occurred, we tracked the amount of contexts created as a result of context splitting, and each website had several hundred contexts. In this case, they did not yield any benefit, as the call-graph edges identified were identical, but they did not appear to impact the performance of the analysis or its correctness.

Now that we have established that our implementation does not empirically appear to affect correctness, we can consider the question of whether it correctly performs context splitting. We manually authored a suite of test programs targeted at situations where a traditional approach simply does not contain enough contextual information to properly identify edges that cannot truly occur, but where context splitting in the right place would help. These contain several limitations that may not extend to real-world programs. They are targeted at the particular use cases that our context splitters are expected to handle, and they are of course far smaller-scale than real-world programs. Still, they demonstrate that our implementation can pick up on the context sensitivity it is designed to. In Section 3.5.1.2, we will discuss our experiments demonstrating that these context splitters provide benefit in the real world.

In the process of performing these tests, especially the hand-authored test programs, we uncovered several issues in the underlying WALA code. We promptly communicated these to the WALA developers, and they were able to fix the bugs with the help of our test cases.

3.5.1.2 Large-Scale Experimental Setup

To assess whether our approach provides any empirical benefit, we sought out the publicly available list of Alexa Top 50 websites,[1] both in the US and globally. Because many of these websites change over time, and some perform A/B testing[19] (presenting one version of the website to some users, and a different version to others, in order to track the effectiveness of a new layout), we archived versions of these websites so that we could ensure that each analysis operated over identical JavaScript code. Our versions of the Top 50 Global websites and the top 50 US websites are archived as of 09/12/2019, which is the same date we accessed the Alexa Top 50 listings. Some sites were excluded due to containing adult content, some were excluded because WALA cannot correctly parse them (and thus no analysis could be performed on them using WALA), and some were excluded because they were too similar to other websites already on the list (i.e., google.com vs google.co.jp). Of those with parsing errors, we notified the WALA developers of these issues, and they were able to resolve some of them, but some are the result of newer, more lax, versions of JavaScript than WALA is currently equipped to handle, and some are the result of syntactic errors in the JavaScript code. We also added 2 variants of the popular JavaScript library, jQuery (version 3.4.1), one that has been obfuscated/minimized, and one that has not. While not a website, it is a necessary pre-requisite for properly computing a full call-graph for many real-world websites. In the end, we arrived at a curated list of 49 websites to perform our experiments on. The websites span a large spectrum of use cases, from search engines, to banking websites, streaming websites, and forums.

Next, we settled upon a number of configurations to test. For each website, we ran the following configurations:

1. Standard WALA, including the correlation tracking heuristic discussed in Section 3.3.
2. Standard WALA, but with correlation tracking disabled.
3. Each possible combination of context splitters and split range policies. This includes each context splitter with each range policy, as well as all combinations of combined policies in all possible orders.

[1] `https://www.alexa.com/`

There are 9 individual context selectors (3 split ranges * 3 splitters). When it comes to combining policies, we restricted ourselves to only use each splitter twice, as it is contradictory to specify each multiple times. Thus for pairs of context selectors, there are 9 options for the first item in the pair, and 6 for the second item (as 3 of the 9 individual policies would use the same context splitter). This is a total of $9 * 6 = 54$ possible pairs. For triples, take the 54 pairs and add one policy. Since two context selectors have already been specified, the only possible option for the third element of the pair is the remaining one, with any of the 3 range policies: thus, 3 possible choices for the last element in the triple. Therefore there are $3 * 54 = 162$ triples.

All told, there are $9 + 54 + 162 = 225$ combinations of context splitting using the 3 context splitters and 3 range policies described in Section 3.4.2 and Section 3.4.3. We disabled WALA's version of correlation tracking when running these configurations.

Obviously, this is a sizeable number of configurations to test, a total of 11,123 individual analysis runs ($49 * 225 + 49 * 2 = 11{,}123$). It was not feasible to manually examine the output of each, nor to run them on a single machine. We used HTCondor [22] to run these jobs in parallel, on machines of comparable power. We requested 500MB of storage on each machine, and 13.5 GB of memory for each. If any jobs ran out of memory, we requested twice the memory up to two times, limiting our memory usage to a maximum of 54GB. Each job was run in a Unix-based Docker container [4], to ensure as similar an environment as possible. For all external dependencies, we fetched the exact same versions, to prevent a mismatch in version number causing unfair differences between analysis runs.

We gave each job a 5 hour window (measured in CPU-time) to complete, upon which we notified the job and attempted to halt it gracefully, dumping any output if it was in the process of outputting results. After 2 further CPU-hours, we terminated the job. This proved ample time for all of the jobs that did manage to produce a call-graph: the longest-running successful job ran a total of $21,652.41$ CPU-seconds, which is roughly 6 hours, well within our grace period. Most jobs ($\approx 50\%$) finish in under 30 minutes.

3.5.2 Results

We recorded the CPU time, wall-clock time, and memory usage of each job. Unfortunately, these numbers vary too much depending on the website to be adequately summarized per-strategy. In fact, for most strategies, the standard deviation of the CPU time used was the same order of magnitude as the CPU time itself. This actually makes sense, as we have a

Table 3.1: WALA vs. Context Splitting

	Configuration	Finished count	Websites Unfinished
WALA	Correlation on	36	amazon, bankofamerica, ebay, espn, hulu, imdb, providr, taobao, tmall, tumblr, twitch.tv, yahoo, yelp
	Correlation off	43	jquery, login.tmall, ebay, imdb, nytimes, tumblr
Best Splitting	(FR, MU), (PN, MU), (CT, TM).	49	None
	(FR, MU), (PN, MU), (CT, MU).	49	None
	(PN, MU), (CT, TM), (FR, TM).	49	None
	(FR, TM), (CT, MU). Either order.	49	None
	(FR, MU), (CT, TM). Either order.	49	None
	(PN, MU), (FR, MU), (CT, TM). Latter two in either order.	49	None
Worst Splitting	(PN, DR), (FR, MU), (CT, DR).	37	alipay, amazon, baidu, blogspot, espn, hulu, nytimes, providr, tumblr, twitch, vk, zillow
	(CT, DR), (FR, MU), (PN, DR).	37	jquery, amazon, bankofamerica, espn, hulu, imgur, nytimes, providr, tumblr, twitch.tv, vk, zillow
	(CT, MU), (FR, MU), (PN, DR).	37	jquery, jquery-min, amazon, espn, hulu, nytimes, ok.ru, providr, tumblr, twitch.tv, vk, zillow
	(PN, DR), (FR, MU), (CT, TM).	36	jquery, jquery-min, alipay, amazon, espn, hulu, jd, nytimes, providr, tumblr, twitch.tv, vk, zillow
	(PN, DR), (CT, TM), (FR, MU).	36	jquery, alipay, amazon, bing, espn, hulu, jd, nytimes, providr, quora, twitch.tv, vk, zillow

All websites are .com unless otherwise specified. All context splitting configurations listed in order from lowest priority to highest priority. Order as listed unless the row specifies that order does not matter. Field read splitter is abbreviated FR, phi node splitter is abbreviated PN, and correlation tracking splitter is abbreviated CT. Must-use range is abbreviated MU, transitive must-use range is abbreviated TM, and dominators range is abbreviated DR. Therefore the first splitting policy should be interpreted as the field read and phi node splitters using the must-use range and the correlation tracking splitter with the transitive must-use range.

diverse array of JavaScript code among our 49 websites. Some websites contain minimal JavaScript code, almost none, while others contain a sizable amount of JavaScript. The same holds true for memory usage. Additionally, while we took every measure we could to ensure our runtime environments were as similar as possible, HTCondor cannot guarantee that jobs are run with a similar workload on the physical device being used to run each job, and not all machines had the same hardware. Due to the sheer volume of jobs, we were unable to record the call-graphs produced by each job, and therefore do not compare their accuracy, only the quantity of websites they were able to successfully analyze.

For that reason, we will focus on comparing the number and names of finished websites with each strategy. Table 3.1 compares the two WALA configurations we tested with our context splitting approach. Of course, we are unable to present all of the context splitting configurations given the sheer quantity. We instead present some of the best and some of the worst context splitters. Note that 3 of the context splitting configurations listed actually represent multiple possible orderings. (As a reminder, the ordering for context splitting only matters when multiple splitters would choose the same split site with *different* split ranges.)

The first thing to compare is the number of websites that finished call-graph construction. Notice the difference between WALA with and without correlation tracking. Perhaps surprisingly, WALA performs better without correlation tracking. However, notice the list of websites that were unable to finish: the only one they have in common is tumblr. We looked into the issue here, and it turns out that correlation tracking sometimes fails to extract the `for-in` loop body into a new method, and often halts with an error when this occurs, which of course counts as an unfinished website result. When it works, it does enable WALA to analyze several websites that otherwise proved impossible. In fact, if we take the union of the websites they finish, WALA could finish 48 of the 49 websites we looked at.

When context splitting is at its best, it is able to analyze all 49 websites. 9 configurations are able to tackle all the websites in our test suite, and 10 can finish 48 websites (the latter omitted from Table 3.1 for brevity). Even at its worst, it performs similarly to WALA with its version of context splitting. There are a few patterns in context splitting that may prove interesting to explore. First, notice that none of the best *or* the worst context splitting configurations involve only a single context splitter. We will discuss singular context splitters more when examining Table 3.2, but for now it is interesting to note that they do not tend to stand out. Another interesting data point is that the context splitting implementation of correlation tracking appears to be necessary in order to analyze all 49 websites. All of our best context splitters use correlation tracking. But correlation tracking on its own does not appear

sufficient, either. Phi node splitting does not appear to hinder completing all 49 websites, but it also does not seem to provide much advantage, since correlation tracking and field read splitting finish all websites, but correlation tracking and phi node splitting cannot.

Next, take a closer look at the split ranges. The most important piece to highlight here is where the dominators range shows up. . . and where it does not. The dominators range occurs in *none* of the best configurations, and in *all* of the worst configurations. As our broadest range, this makes some sense – perhaps the additional overhead of maintaining context sensitivity over a larger range simply does not provide sufficient benefit.

As for the websites each strategy cannot finish, there is a lot of commonality among the worst context splitters. This makes sense as well, as the strategies themselves are not very distinct. Each uses all three policies, each uses the dominators range at least once, and while the order of each does vary, the dominators range does seem to be highest priority quite often.

To separate out the performance of each context splitter, Table 3.2 contains the number of finished websites using each individual context splitter and each range. Overall, the correlation tracking splitter tends to perform best, while the field read splitter performed worst. The one surprise was the phi node splitter, which manages to analyze 47 of 49 websites when given the must-use range policy. Despite its excellent solo performance, it seems likely that it adds too many split points that do not provide sufficient benefit, especially in tandem with another context splitter. This may explain why the must-use range performs so much better than the other range policies with the phi node splitter.

While the field read splitter does not perform nearly as well as the correlation tracking splitter, it complements the correlation tracking splitter especially well. The field read splitter can handle all of the websites the correlation tracking splitter cannot. That explains why they work so well jointly.

3.5.3 Summary

Overall, both sets of experiments together demonstrate that context splitting provides substantial benefit over an approach with context sensitivity only at the level of the function. Even in the worst case, context splitting does not appear to make the odds of analyzing any given website much worse.

Due to the improvement demonstrated here, the WALA developers have agreed to incorporate the context splitting framework into their open-source project. This will allow a greater number of developers to benefit from this technique without changing the platform they use to construct call-graphs.

Table 3.2: Single Context Splitters

Configuration	Finished count	Websites Unfinished
(FR, DR)	43	t.co, baidu, espn, nytimes, tumblr, twitch.tv
(FR, MU)	43	jquery, alipay, craigslist.org, espn, hulu, quora
(FR, TM)	44	alipay, baidu, espn, jd, quora
(PN, DR)	42	baidu, bing, espn, imdb, ok.ru, providr, twitch.tv
(PN, MU)	47	imdb, ok.ru
(PN, TM)	43	alipay, baidu, craigslist, espn, imdb, nytimes
(CT, DR)	46	bankofamerica, imdb, live
(CT, MU)	46	diply, imdb, zillow
(CT, TM)	44	apple, bankofamerica, blogspot, diply, imdb

All websites are .com unless otherwise specified. Field read splitter is abbreviated FR, phi node splitter is abbreviated PN, and correlation tracking splitter is abbreviated CT. Must-use range is abbreviated MU, transitive must-use range is abbreviated TM, and dominators range is abbreviated DR.

Based on the empirical data, the most general purpose context selectors seem to contain the correlation tracking splitter and the field read splitter and do not use the dominators range. If a developer has a particular sense of the sources of ambiguity present in their program, they may opt to choose their configuration based on their inside knowledge. Future researchers can use our API to construct their own context selectors, should evidence arise of new sources of imprecision.

3.6 Future Work

To reduce the amount of analysis, it may be possible to also perform an "additive" split, which would maintain separate copies of the function's IR in both split ranges rather than nesting them. Instead of exploring each combination of possible values from the split points, each split would be performed independently. This would result in analyzing shared instructions $n + m$ times, but would potentially lose some precision.

This work compared hand-created policies based on an intuitive sense of where state explosions tend to occur, and how to prevent them. Future work could extend this by empirically studying where state explosions occur and designing policies to address those particular program points. Alternatively, a machine learning analysis could direct the creation of policies based on call-graph construction time and precision for real-world websites.

This work only addresses contextual information within a single method. It is possible that it may be useful to retain contextual information between methods, in particular with

contextual information about globals. Future work could explore how to further extend context splitting to handle cross-method contextual information.

WALA has a number of options for running, many of which are heuristics designed to sacrifice correctness for completion. It would be useful to compare how well context splitting interfaces with each of them, and whether the extra precision added by context splitting can help counteract the imprecision introduced by heuristics.

While the number of websites completed using context splitting is impressive, they still take more time than developers may be prepared to allow, depending on the task at hand. Future work could explore ways to restrict the split sites to smaller sets, or focus on a larger quantity of context splitters that each create a far smaller number of split sites. These context selectors could do more sophisticated logic, perhaps contingent on how the values are used in the future, rather than simply pattern matching on the source of the value.

So far, the framework we have built applies the same range policy to each split site depending on what context splitter was used. Further work could experiment with instead determining the appropriate split range for a split site based on features of the call-graph and dataflow.

3.7 Stronger Together

We hope that our approach will not be in contrast to other solutions to the large problem of JavaScript static analysis, but rather, serve as a locus of cooperation. Because the primary issue is effectively a points-to analysis, many avenues for reducing ambiguity work extremely well together, improving the results of both analyses. Likely no single analysis technique will ever be fully able to address the breadth of all the complications of analyzing JavaScript code. Instead, many techniques must work together in the same analysis framework in order to combat the complexity of JavaScript.

We would like to highlight the way few techniques may prove symbiotic to context splitting, though these are by no means the only such techniques. As discussed in Section 3.3, one area of ongoing work is that of recovering the possible values of strings [16, 34, 39, 40], particularly those passed into `eval` statements. Some approaches focus on a traditional static points-to analysis, while others focus on a symbolic execution approach. While the aim is different, the tasks are similar enough to impact one another. The more precisely we understand these string values, the more precise the analysis can be about edges eval statements may or may not introduce, or updates to state information that it may perform, which allows us to reduce the

number of possible call targets at some call sites. Likewise, the more precise the call-graph becomes, the less ambiguity there will be for string values. Improving either analysis improves both, as they are interdependent in nature. Another avenue of future collaboration lies in the area of incremental typing analyses, which use optional type annotations when provided, as well as a static analysis, to provide as much type information as possible, which can augment call-graph and other points-to-related analyses. Reducing the state explosion caused by an imprecise call-graph can reduce the number of possible types for objects, while at the same time, improved typing information may further restrict call-site targets.

Because context splitting can be easily layered over other static analysis techniques without requiring extensive modifications to the underlying analysis engine, it allows cooperation between these analysis strategies with minimal effort (as long as the strategies themselves are implemented on the same engine). Big problems require many minds and many tools working together in harmony in order to succeed.

3.8 Conclusion

JavaScript call-graph development has a long way to go. Due to the way that any imprecision can spiral and lead to the algorithm completely stalling, any and all tools working towards the end of improving precision need to work together. Context splitting opens up a new space to explore in curbing the imprecision that JavaScript tends to create. Context splitting has the potential to drastically improve the performance of JavaScript call-graph analysis algorithms. Our experiments with WALA have demonstrated that context splitting provides much-needed clarity to reduce the state space and prevent the analysis from becoming infeasibly expensive in terms of time and memory. While there are still many promising directions to explore, our novel technique performs better than the standard techniques on real-world JavaScript programs currently powering well-known websites. The flexibility to craft context splitters and split range policies based on the needs of a particular program will allow developer need to drive decisions made by call-graph constructions. Its placement in WALA will allow it to be accessible to developers who are already on the hunt for a call-graph construction algorithm that can successfully analyze their code.

Part IV

Conclusions

4 CONCLUSIONS

This dissertation has laid out our work addressing areas where type information for tool support is both scarce and most critical. In both domains that we focused on, we have provided novel alternatives to the traditional approaches that outperform existing techniques and alleviate developer frustration.

Of course, there is still much work to do to provide tools for such domains: it is as impossible to address *all* these domains in the way that it is impossible to read every book ever written. Not only is the sheer number overwhelming, more constantly crop up as computer science advances. The goal of this work has never been to address every such problem, nor even to address every problem in the two domains we focused on. Even so, we wish to draw attention to future avenues for improvement, both in the areas we directly address, and in other areas we are aware of, but have not (yet) had the ability to explore. A discussion of these possible future areas of research are laid out in Section 4.1.

Section 4.2 is included for other program analysts who find themselves facing similar challenges we did when it comes to evaluating their tool. When exploring an under-researched topic, quite often it is difficult to know exactly how to compare one's work to existing work, or how to gauge its effectiveness. Section 4.2 discusses our experience, recounts how we addressed concerns about evaluation, and presents our advice for future program analysts. Finally, Section 4.3 recaps our accomplishments, addresses the current space of program analysis for domains lacking type information, and concludes.

4.1 Future Work

Of course, the area of program analysis in the face of limited type information is vast, and no one work can fully encompass it. To that end, we wanted to highlight some specific areas that deserve particular attention moving forward, in hopes of better equipping developers with the tools they need to write and maintain well-written code. While this section focuses in particular on areas related to those discussed in this dissertation, we urge readers to keep an eye out for other areas where developer tool support is neglected due to limited type information. More often than not, this section will contain questions rather than answers. In our experience, a good research project begins with a question. "Why," "how," and "what" are both the most useful tools for a researcher to have, and the most frustrating ones.

Our array length inference technique discussed in Chapter 2 makes an important step forward in automating the creation of language bindings to C libraries, but more work remains. The first step forward is likely conducting a survey among polyglot developers, to determine what the current pain points are. What parts of connecting two languages together might put developers off even attempting? And are there ways that tools can make that process less painstakingly difficult?

It may also be helpful to explore more different conventions programmer use in different languages (such as how array-length information is represented for C programs), and identify places where developer intention can be leveraged to make better heuristics for program analysis in difficult spaces. During our research, we discovered that C developers frequently use pointers in a way that indicates their intended use in the data itself. One example is discussed in Section 2.5, but because we were specifically looking for length information, there may be all sorts of other predicated types that we simply did not come across. With the limited type information baked into C pointers, developers have learned to be very creative with it, and unfortunately, this can introduce a barrier when connecting C to other languages, as other languages have more structured pointer types. Are there other kinds of predicated types in common use? What do developers currently do to match predicated types to types in languages other than C? How can this be partly or entirely automated, to remove the burden from developers who just want to use a C library without any fuss?

In Chapter 3, we discussed our flexible, context-sensitive approach to constructing JavaScript call-graphs. While our technique provides call-graphs for a number of JavaScript programs that have previously defied analysis, there remains much work to be done in this realm to truly equip JavaScript developers with the same tool support as those in statically-typed languages.

We have identified some sources of runtime explosion, but many more remain to be discovered. We believe a machine learning technique may be especially helpful here, though certainly many questions need to be answered in order to properly train an algorithm for deciding the appropriate amount of context sensitivity to apply. What should the features be? How should we compare two algorithms if neither succeeds in producing a call-graph in a timely fashion? Can one be "better" than another if neither finishes? Call-graph-construction techniques are iterative, but the number of iterations is unbounded (though theoretically finite); they do not come with a progress bar.

Our experience indicates that generally a call-graph-construction algorithm experiences something of a snowball effect when imprecision creeps into the call-graph under construction.

The more unnecessary edges are in the graph, the bigger the state that needs to get explored grows, and the greater the number of iterations that need to be made. Worse, all of those unneeded edges don't stop there: generally, they continue creating more and more edges, none of which can actually exist. Thus, one very promising candidate for a metric in this space is whichever partially finished call-graph has the fewest edges. Of course, to use this metric, one has to have some basic assumptions about the way in which the algorithms function: monotonicity, that an algorithm encountering an error will not just stop but will report it, etc. There also will likely need to be a way to temper this value: another possible explanation for two call-graphs being vastly larger is that the algorithm producing it was simply more efficient and got closer to the true call-graph.

One benefit of the call-graphs we produce is that they contain embedded context-sensitive information used to create them. This could potentially be of great use to static analyses using these call-graphs. We leverage this information to determine call targets, but this is really a more general technique that benefits the pointer analysis as well. Static analysts who want a call-graph, then, could take advantage of this contextual information to improve their analysis.

4.2 Advice for Program Analysts

In the course of constructing our program analysis techniques, we ran into several hurdles that we believe are both common and not frequently discussed. We provide these notes for readers who may find themselves in a similar situation to us, who are working in a challenging space where benchmarks and ground truth simply do not exist yet. This section will lay out our experiences and how we resolved to handle them, and each subsection wraps up with a discussion of our recommendation for program analysts in similar circumstances.

4.2.1 Notes on Evaluating Without Benchmarks

In both Chapter 2 and Chapter 3, we found ourselves unexpectedly without a set of benchmarks to compare our results and determine whether our tool was effective or not. This can make a rigorous analysis difficult, since it is unclear whether any particular analysis is sufficient to show the success of an approach. But sometimes, there is no standardized benchmark to use, whether because the problem is sufficiently novel, or because existing work does not agree upon a standardized set of test cases. In those cases, it is even more important than usual to demonstrate the efficacy of your approach, even though how to do so is less clear than usual. So, how *does* one handle such a lack of experimental benchmarks?

In the case of Chapter 2, no previous tool had been written to produce these annotations, so there was no standard suite of benchmark code to compare our approach on. But because the annotations were already in use by many users, there *were* several large-scale, real-world programs with human-created annotations. As discussed in Section 2.4.3.2, these annotations were not perfect, but they provided excellent guidance as to a set of programs to use. We couldn't treat these annotations as ground truth (see Section 4.2.2), but we ensured that the bulk of our experiments were run on programs with hand-authored annotations, since these are programs that users took the time to craft annotations for. We could have chosen any set of C programs, but that space is incredibly vast, and we specifically wanted to look for library code that makes heavy use of arrays and/or strings. Code with these annotations already in place was much more likely to make use of arrays and strings, and by definition was library code rather than a complete program. This allowed us to narrow our search, but we made certain to also include in our analysis a few experimental candidates that did not have annotations at the time. Adding some programs without existing annotations to our experimental suite enabled us to verify that our approach generalized to datasets not written with annotations in mind.

With Chapter 3, while other tools existed, there simply was no existing agreed-upon set of benchmarks: various tools used different tests, and not all were entire programs. We tried several approaches before arriving at the one described in Section 3.5. We knew we wanted a large-scale set of tests on real-world code, and we knew we wanted that code to be entire program code, not library code. (In Chapter 2, we intentionally sought out library code because it was more challenging, and also more important, to have length annotations for arrays in library code rather than entire program code; in Chapter 3, the more difficult analysis is of whole program code, where the additional code provides more scalability challenges.) Because we based our approach on an existing framework (WALA: see Chapter 3 for details), we had a natural way to compare our approach to that of WALA's standard analysis. Unfortunately, incorporating other tools' results proved to be too time-consuming, but comparing the difference between WALA with and without our changes helps indicate the improvement our analysis provided. Given that the general process for constructing a call-graph is similar at a very high level, it seems likely that these gains would also apply to other call-graph construction techniques, not simply WALA's. A few complications that we considered when choosing our test base:

- If we simply chose our own programs, we ran the risk of accidentally selecting programs that aligned well with our assumptions, causing our results not to generalize to real-world developers. Thus, we settled on using the Amazon Alexa listing of the top-100 US and

Global websites, which uses website traffic to determine the most "popular" websites. This has the advantage of being far more objective than simply selecting websites the authors like, and reflects real-world websites that have prospered.

- Yet, at the same time, most of these real-world websites do not have open-source versions. Many of them are heavily obfuscated or minimized, techniques that make the code far less readable. Of course, non-human-readable code is perfectly valid to analyze, but determining correctness of our analysis becomes virtually impossible. Knowing whether our analysis removes any edges that ought to exist in the call-graph is an incredibly important question to determine whether our analysis makes an otherwise-correct call-graph construction algorithm incorrect. Because we do not have ground truth (see Section 4.2.2), we cannot simply compare our result to the correct answer. Therefore, we found a secondary compilation of websites, primarily ones originally written in the 1990s. We manually examined this list and distilled it down to 7 websites from the original list that were human-readable, reasonably complex (i.e., contained at least 200 lines of non-trivial JavaScript), but simple enough that both standard WALA and our context splitting approach finished in under 10 minutes of analysis time. The websites used varied greatly in scope and purpose. After obtaining call-graphs using both WALA's analysis approach, and WALA + context splitting, we sorted the edges in the call-graphs and compared them using a diff. In every case, the context-splitting approach was verified to be identical to standard WALA's call-graph.

- We could not directly fetch the website anew each time we ran a test, due to the potential for A/B testing. This was relatively easily resolved: we simply cached existing versions of the websites. For those doing analysis on websites, we recommend caching any websites used for analysis if using live, production versions, so as to guarantee the exact same website is analyzed each time.

In general, we advise following the general guidelines below when you must select your own set of benchmarks, rather than use existing ones. Of course, if existing benchmarks exist and are compatible with your analysis, this is to be preferred, since it will allow your experimental results to be more directly compared to others' results.

1. Note versions, release dates/times, and the original input files wherever possible so that future scientists can reproduce your results exactly and compare their results to yours.

2. Select inputs that are as objective and as similar to the intended use case as possible.

3. If necessary, verify your results using a separate dataset, and take care to select this dataset as objectively as possible.

4. Ensure that your inputs are sufficiently complex and diverse to generalize.

5. Make use of public information about popularity of inputs, where possible.

4.2.2 Notes on Evaluating Without Ground Truth

In both Chapter 2 and Chapter 3, we found ourselves without a true gold standard to compare our results against. This complicates evaluation, as not only did we lack a standard benchmark (see Section 4.2.1), we also lacked the ability to easily determine whether our analyses produced incorrect results.

When attempting to evaluate our array-length-inference analysis (see Chapter 2), the closest we had to ground truth was human-emitted annotations. As discussed in Section 2.4.3.2, these annotations were incorrect a surprising amount of the time, so they could not truly be relied upon as ground truth. Instead, we (rather painstakingly) manually compared times when our algorithm and the human annotators disagreed. Fortunately for us, the number of differences was not as large as it could be. Naturally, this does not give us a perfect metric: there may be plenty of places where both human authors and our analysis were incorrect in the same way. However, because our strategy was decidedly different from that of human authors (who primarily used documentation and variable names, while our analysis ignores these), this seems reasonably unlikely to occur at scale.

During the work covered in Chapter 2, finding ground truth at the scale of real-world programs was practically impossible. Many of the programs we wished to analyze had not had any analysis successfully complete previously, so there was no way to compare the two sets of call-graphs. And the programs were generally complex enough that constructing call-graphs by hand was not feasible, either. Thus, we used a series of smaller, hand-authored programs to verify whether the call-graphs produced exhibited known problems (for example, a portion of the code that does not exist in the final call-graph), and also a series of human-readable programs that were simple enough that we could produce call-graphs using an existing technique and compare the two. We found no issues with the call-graphs produced, and verified that they were identical to those produced by the unmodified version of WALA. This allowed us to gain confidence in our results on larger programs, where we did not have ground truth, and the original code was difficult or impossible to manually verify. Effectively, we were attempting to demonstrate that we had followed what we dubbed the "*Hippocratic*

Oath for Program Analyses:" an analysis based on an existing analysis should not make the result less correct. (Of course, correctness itself can be a tricky concept, but for our purposes, "less correct" means that the output provides misleading or false information, not simply overapproximations of the truth.) We also pondered potential metrics for comparing partially-completed call-graphs, discussed further in Section 4.1: these are intended for use not only when ground truth is unavailable, but also to predict which algorithm is "more scalable" when neither algorithm is able to finish.

In general, the first step is to look at what developers do when faced with this problem. In most cases, even if there is not a "state-of-the-art," people must be getting around the lack of tool support in some way, even if that way is manual rather than automated. Search for places where you can compare the results of your technique to the results of the existing workaround or state-of-the-art. Even if they may not scale to the degree that yours does, first ascertain that you outperform or match them on smaller inputs. If necessary, manually compare output on a random subset of results. Then, when you test at scale, verify some of your results (selected at random) manually, even if only a subset.

4.3 Conclusion

Our tools were designed with the developer in mind, to alleviate frustration and improve the code that developers write, even in spaces where limited type information complicates this. Although this space certainly has room for future research, our contributions address real and present problems for developers in the realm of polyglot development and JavaScript development, both of which are growing in popularity.

In the polyglot domain, 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. In many cases, we found our results to be more reliable than human annotators, who relied on heuristics that prove unexpectedly faulty (such as the name

of a method or variable). Our results perform admirably on both code designed for these annotations, and code that was not written with such annotation in mind.

JavaScript call-graph analysis is also complicated by a quirk of its type system, its object model, which only encourages properties to be added on-the-fly, as opposed to a static class hierarchy. In many real-world JavaScript programs, call-graph construction has been frustratingly out of reach, making it difficult for JavaScript developers to find static program analysis support. To address this issue, we constructed a novel, flexible, context-sensitive technique (context splitting) to be applied alongside existing call-graph construction techniques. Generally, context sensitivity in JavaScript analysis is provided at the granularity of the function. With our approach, context sensitivity can be applied to a finer granularity, allowing users to identify sources of imprecision in the call-graph and the portions of the code in which context sensitivity is needed. We have implemented several policies for context splitting (predefined for convenience and testing), but users can also define arbitrary policies. Our technique provides significant improvement in call-graph construction without harming correctness of the resulting graph. We were able to analyze all 49 of the real-world, popular websites we considered, while the underlying call-graph construction algorithm we used, WALA, could analyze only 43. Our implementation is expected to be incorporated into a future release of WALA, and can be used in conjunction with existing call-graph analysis techniques without interfering with them and without modifying the call-graph.

Developers who are not given the tools to succeed struggle to find workarounds and often simply find another way when they do not have the resources they need to do things the right way. By addressing two significant and popular areas where tool support is lacking, we hope to empower developers to write better, more reliable code with far less frustration.

Part V

Appendices

A GITHUB DATA

Two Sigma Ventures maintains a list of the top 100 GitHub projects [47], updated daily. We visited each website and recorded the list of languages by prevalence in the project, as reported by GitHub. This list is ordered by popularity score, provided by Two Sigma Ventures. The list is included in Table A.1 for completeness.

Languages that report "0.0%" prevalence do so because the percentage is less than 0.1%, and thus rounded to 0 by GitHub. Thus their prevalence is higher than the reported 0%, but very minimal. Entries with "Other" languages reported consist of either several different languages comprising very small amounts of the project, or represent files with extensions that GitHub was not able to recognize and map to a language.

Note that very few projects (5%) report only a single language, and the overwhelming majority (73%) report at least 3 languages, with an average of 4.5 languages used per project. While some of the "languages" reported actually amount to build scripts (`Makefile`, `CMake`, etc), generally they represent a very small portion of each project when they are present. There are also some languages commonly paired together, such as `JavaScript`, `HTML`, and `CSS`, or variations on similar languages like `JavaScript`, `Typescript` and `CoffeeScript`; `CSS` and `SCSS`; or `C++` and `C`. Despite this, it seems clear that a significant percentage of popular, current open-source projects use more than one programming language.

9% of the projects use `C` in some capacity. Of those, `redis` is the only project primarily comprised of `C`: for projects using `C`, the average percent of the project developed with `C` is 15.4%. If `redis` is ignored as an outlier, this average drops to just 7.2%. This indicates that while `C` is still an important development language in modern projects, it serves as a secondary, rather than a primary language, for most projects. JavaScript appears in 32 projects, and averages 46.1% of projects it is used in, indicating it is far more often used as a primary development language.

Table A.1: List of top 100 GitHub Projects [47] as of 06/07/2021

Project	Languages
https://github.com/twbs/bootstrap	JavaScript: 41.7% HTML: 31.6% SCSS: 14.4% CSS: 12.2% PowerShell: 0.1%
https://github.com/tensorflow/tensorflow	C++: 61.6% Python: 25.6% MLIR: 3.1% Starlark: 2.9% HTML: 2.7% Go: 1.2% Other: 2.9%
https://github.com/twbs/bootstrap	JavaScript: 41.7% HTML: 31.6% SCSS: 14.4% CSS: 12.2% PowerShell: 0.1%
https://github.com/facebook/react	JavaScript: 95.1% HTML: 2.0% CSS: 1.2% C++: 0.8% TypeScript: 0.4% CoffeeScript: 0.3% Other: 0.2%
https://github.com/vuejs/vue	JavaScript: 97.6% Other: 2.4%

Table A.1 continued from previous page

Project	Languages
https://github.com/kubernetes/kubernetes	Go: 96.6% Shell: 2.9% PowerShell: 0.3% Makefile: 0.1% Dockerfile: 0.1% Python: 0.0%
https://github.com/angular/angular.js/	JavaScript: 98.1% HTML: 1.2% Other: 0.7%
https://github.com/apple/swift	C++: 51.2% Swift: 44.2% Python: 2.3% CMake: 0.7% Objective-C: 0.5% C: 0.4% Other: 0.7%
https://github.com/moby/moby	Go: 96.6% Shell: 2.0% PowerShell: 0.9% Dockerfile: 0.2% Makefile: 0.2% Python: 0.1%
https://github.com/angular/angular	TypeScript: 82.6% JavaScript: 12.9% Starlark: 1.6% HTML: 1.5% CSS: 0.8% SCSS: 0.3% Other: 0.3%

Table A.1 continued from previous page

Project	Languages
https://github.com/spring-projects/spring-framework	Java: 98.5% Kotlin: 1.2% Groovy: 0.1% AspectJ: 0.1% FreeMarker: 0.1% CSS: 0.0%
https://github.com/microsoft/vscode	TypeScript: 94.0% JavaScript: 3.3% CSS: 1.5% Inno Setup: 0.7% HTML: 0.4% Shell: 0.1%
https://github.com/spring-projects/spring-boot	Java: 98.9% HTML: 0.3% Kotlin: 0.3% Shell 0.2% JavaScript: 0.2% Groovy: 0.1%
https://github.com/rails/rails	Ruby: 95.4% HTML: 3.2% JavaScript: 0.9% CSS: 0.3% CoffeeScript: 0.2% Shell: 0.0%
https://github.com/ohmyzsh/ohmyzsh	Shell: 98.7% Other: 1.3%
https://github.com/microsoft/TypeScript	TypeScript: 100.0%

Table A.1 continued from previous page

Project	Languages
https://github.com/symfony/symfony	PHP: 98.8% Twig: 1.0% CSS: 0.1% JavaScript: 0.1% HTML: 0.0% Shell: 0.0%
https://github.com/mrdoob/three.js/	JavaScript: 99.2% Other: 0.8%
https://github.com/facebook/create-react-app	JavaScript: 98.3% Shell: 1.3% CSS: 0.1% HTML: 0.1% AppleScript: 0.1% TypeScript: 0.1%
https://github.com/apache/dubbo	Java: 99.5% Other: 0.5%
https://github.com/scikit-learn/scikit-learn	Python: 97.8% C++: 1.4% Other: 0.8%
https://github.com/webpack/webpack	JavaScript: 99.9% WebAssembly: 0.1% CSS: 0.0% CoffeeScript: 0.0% HTML: 0.0% Less: 0.0%

Table A.1 continued from previous page

Project	Languages
https://github.com/angular/angular-cli	TypeScript: 95.7% Starlark: 1.8% JavaScript: 1.6% EJS: 0.5% HTML: 0.2% Shell: 0.2%
https://github.com/jquery/jquery	JavaScript: 93.8% HTML: 5.3% Other: 0.9%
https://github.com/dotnet/aspnetcore	C#: 87.6% JavaScript: 3.9% C++: 2.8% HTML: 2.4% TypeScript: 1.3% Java: 0.8% Other: 1.2%
https://github.com/redis/redis	C: 81.4% Tcl: 17.4% Ruby: 0.4% Shell: 0.4% Makefile: 0.3% C++: 0.1%
https://github.com/denoland/deno	Rust: 59.7% JavaScript: 21.2% TypeScript: 19.0% Other: 0.1%

Table A.1 continued from previous page

Project	Languages
https://github.com/ant-design/ant-design	TypeScript: 44.3% JavaScript: 31.1% Less: 24.4% Other: 0.2%
https://github.com/pallets/flask	Python: 99.9% Other: 0.1%
https://github.com/apache/superset	Python: 48.8% TypeScript: 27.5% JavaScript: 19.6% HTML: 1.4% Less: 1.3% Shell: 1.2% Other: 0.2%
https://github.com/photonstorm/phaser	JavaScript: 99.6% Other: 0.4%
https://github.com/ansible/ansible	Python: 88.2% PowerShell: 7.4% Shell: 2.0% C#: 1.9% Jinja: 0.4% Makefile: 0.1%
https://github.com/laravel/framework	PHP: 99.2% Other: 0.8%

Table A.1 continued from previous page

Project	Languages
https://github.com/bitcoin/bitcoin	C++: 66.6% Python: 18.9% C: 9.2% M4: 1.6% Shell: 1.6% Makefile: 1.0% Other: 1.1%
https://github.com/electron/electron	C++: 55.7% TypeScript: 24.7% JavaScript: 6.7% Objective-C++: 6.7% HTML: 3.3% Python: 2.1% Other: 0.8%
https://github.com/SeleniumHQ/selenium	Java: 34.8% C#: 16.8% JavaScript: 14.0% C++: 12.4% HTML: 8.5% Python: 5.1% Other: 8.4%
https://github.com/gohugoio/hugo	Go: 99.1% Other: 0.9%
https://github.com/atom/atom	JavaScript: 88.2% Less: 8.7% CoffeeScript: 3.0% Shell: 0.1% Batchfile: 0.0% Dockerfile: 0.0%

Table A.1 continued from previous page

Project	Languages
https://github.com/apache/spark	Scala: 65.9% Python: 12.1% Jupyter Notebook : 7.5% Java: 7.2% HiveQL: 3.3% R: 2.2% Other: 1.8%
https://github.com/flutter/flutter	Dart: 99.2% Objective-C: 0.2% Java: 0.2% C++: 0.1% Shell: 0.1% CMake: 0.1% Other: 0.1%
https://github.com/PowerShell/PowerShell	C#: 86.0% PowerShell: 12.9% Roff: 0.6% Shell: 0.2% Rich Text Format: 0.1% Dockerfile: 0.1% Other: 0.1%
https://github.com/ApolloAuto/apollo	C++: 83.8% Python: 5.1% Starlark: 4.3% Shell: 3.4% JavaScript: 1.8% Cuda: 1.0% Other: 0.6%

Table A.1 continued from previous page

Project	Languages
https://github.com/pingcap/tidb	Go: 99.7% Other: 0.3%
https://github.com/netty/netty	Java: 98.6% C: 1.1% HTML: 0.1% Shell: 0.1% JavaScript: 0.1% Makefile: 0.0%
https://github.com/animate-css/animate.css	CSS: 70.0% HTML: 18.2% JavaScript: 11.8%
https://github.com/google/guava	Java: 100.0%
https://github.com/ethereum/go-ethereum	Go: 89.1% C: 5.4% JavaScript: 3.5% Assembly: 0.8% Java: 0.2% Sage: 0.2% Other: 0.8%
https://github.com/grafana/grafana	TypeScript: 60.0% Go: 31.0% HTML: 2.2% Rich Text Format: 1.9% JavaScript: 1.9% SCSS: 1.4% Other: 1.6%
https://github.com/foundation/foundation-sites	HTML: 56.6% SCSS: 28.7% JavaScript: 14.7%

Table A.1 continued from previous page

Project	Languages
https://github.com/psf/requests	Python: 99.7% Makefile: 0.3%
https://github.com/x64dbg/x64dbg	C++: 84.9% C: 13.8% QMake: 0.6% CSS: 0.5% Batchfile: 0.2% Assembly: 0.0%
https://github.com/d3/d3	JavaScript: 100.0%
https://github.com/ReactiveX/RxJava	Java: 99.9% Other: 0.1%
https://github.com/apache/incubator-mxnet	C++: 45.9% Python: 36.2% Jupyter Notebook: 8.7% Cuda: 6.0% CMake: 0.9% Shell: 0.7% Other: 1.6%
https://github.com/hashicorp/terraform	Go: 99.3% Other: 0.7%
https://github.com/etcd-io/etcd	Go: 96.0% Shell: 2.4% Jsonnet: 0.8% Makefile: 0.4% Python: 0.2% Dockerfile: 0.1% Other: 0.1%

Table A.1 continued from previous page

Project	Languages
https://github.com/TryGhost/Ghost	JavaScript: 92.3% CSS: 3.5% HTML: 2.4% Handlebars: 1.7% : 0.1%
https://github.com/jekyll/jekyll	Ruby: 71.0% Gherkin: 22.7% JavaScript: 3.9% HTML: 1.2% Shell: 0.8% Dockerfile: 0.2% Other: 0.2%
https://github.com/prometheus/prometheus	Go: 88.7% TypeScript: 6.5% JavaScript: 1.7% HTML: 1.4% Yacc: 0.7% SCSS: 0.3% Other: 0.7%
https://github.com/obsproject/obs-studio	C: 56.9% C++: 37.1% CMake: 2.3% Objective-C++: 1.6% Objective-C: 1.4% Shell: 0.3% Other: 0.4%

Table A.1 continued from previous page

Project	Languages
https://github.com/MaterializeInc/materialize	Rust: 91.4% Python: 4.8% Shell: 1.8% Dockerfile: 0.8% HTML: 0.4% JavaScript: 0.3% Other: 0.5%
https://github.com/pandas-dev/pandas	Python: 95.6% HTML: 2.4% C: 1.9% Shell: 0.1% Smarty: 0.0% Dockerfile: 0.0%
https://github.com/bcit-ci/CodeIgniter	PHP: 97.3% HTML: 1.4% Python: 0.6% JavaScript: 0.3% Makefile: 0.2% CSS: 0.1% Shell: 0.1%
https://github.com/gogs/gogs	Go: 90.6% Less: 4.1% JavaScript: 3.3% Shell: 1.7% Makefile: 0.1% Dockerfile: 0.1% Batchfile: 0.1%

Table A.1 continued from previous page

Project	Languages
https://github.com/apache/kafka	Java: 72.9% Scala: 23.7% Python: 2.9% Shell: 0.3% Roff: 0.1% Batchfile: 0.1%
https://github.com/ElemeFE/element	Vue: 59.1% JavaScript: 26.6% SCSS: 14.0% Other: 0.3%
https://github.com/puppeteer/puppeteer	TypeScript: 72.2% JavaScript: 26.3% HTML: 1.4% Other: 0.1%
https://github.com/h5bp/html5-boilerplate	JavaScript: 80.7% HTML: 19.3%
https://github.com/square/okhttp	Java: 50.0% Kotlin: 49.9% Shell: 0.1%
https://github.com/chartjs/Chart.js	JavaScript: 98.7% Other: 1.3%
https://github.com/AFNetworking/AFNetworking	Objective-C: 96.8% Swift: 1.7% Ruby: 1.2% Other: 0.3%
https://github.com/mui-org/material-ui	JavaScript: 80.0% TypeScript: 20.0%

Table A.1 continued from previous page

Project	Languages
https://github.com/mozilla/pdf.js/	JavaScript: 97.4% CSS: 1.6% Other: 1.0%
https://github.com/Semantic-Org/Semantic-UI	JavaScript: 62.9% CSS: 36.4% HTML: 0.7%
https://github.com/pixijs/pixijs	TypeScript: 72.4% JavaScript: 26.3% Other: 1.3%
https://github.com/adobe/brackets	JavaScript: 83.9% HTML: 13.1% Less: 1.4% CSS: 1.2% PHP: 0.2% Shell: 0.1% Other: 0.1%
https://github.com/axios/axios	JavaScript: 93.3% TypeScript: 3.9% HTML: 2.8%
https://github.com/gin-gonic/gin	Go: 99.6% Makefile: 0.4%
https://github.com/ColorlibHQ/AdminLTE	JavaScript: 77.2% HTML: 16.8% CSS: 4.5% SCSS: 1.5%
https://github.com/expressjs/express	JavaScript: 100.0%
https://github.com/shadowsocks/shadowsocks-windows	C#: 94.5% JavaScript: 5.5%

Table A.1 continued from previous page

Project	Languages
https://github.com/apache/echarts	TypeScript: 90.5% JavaScript: 9.3% Other: 0.2%
https://github.com/gulpjs/gulp	JavaScript: 100.0%
https://github.com/google/material-design-lite	HTML: 36.6% CSS: 34.3% JavaScript: 29.1%
https://github.com/socketio/socket.io	JavaScript: 52.7% TypeScript: 47.3%
https://github.com/PanJiaChen/vue-element-admin	Vue: 69.4% JavaScript: 26.0% SCSS: 4.1% Other: 0.5%
https://github.com/deepfakes/faceswap/	Python: 98.8% Other: 1.2%
https://github.com/vercel/next.js/	JavaScript: 83.5% TypeScript: 16.0% CSS: 0.3% Shell: 0.1% SCSS: 0.1% Dockerfile: 0.0%
https://github.com/ApolloAuto/apollo	C++: 83.8% Python: 5.1% Starlark: 4.3% Shell: 3.4% JavaScript: 1.8% Cuda: 1.0% Other: 0.6%

Table A.1 continued from previous page

Project	Languages
https://github.com/square/retrofit	Java: 96.4% Kotlin: 1.8% HTML: 1.4% Other: 0.4%
https://github.com/grpc/grpc	C++: 44.6% C: 19.7% Python: 12.2% C#: 8.3% Ruby: 3.5% Objective-C: 2.2% Other: 9.5%
https://github.com/ariya/phantomjs	C++: 39.3% JavaScript: 29.4% C: 24.4% Python: 5.8% HTML: 0.7% Shell: 0.2% Other: 0.2%
https://github.com/Alamofire/Alamofire	Swift: 99.9% Ruby: 0.1%
https://github.com/alibaba/fastjson	Java: 99.9% Other: 0.1%
https://github.com/Netflix/Hystrix	Java: 98.5% Clojure: 1.4% Other: 0.1%

Table A.1 continued from previous page

Project	Languages
https://github.com/nwjs/nw.js	JavaScript: 42.9% C++: 34.7% Python: 14.2% HTML: 3.7% Objective-C++: 2.8% Objective-C: 0.6% Other: 1.1%
https://github.com/tesseract-ocr/tesseract	C++: 95.8% C: 1.1% CMake: 1.0% Java: 0.9% Makefile: 0.9% Shell: 0.3%
https://github.com/zxing/zxing	Java: 96.0% HTML: 3.9% CSS: 0.1%
https://github.com/adam-p/markdown-here	JavaScript: 87.2% CSS: 9.5% HTML: 3.3%
https://github.com/necolas/normalize.css/	CSS: 100.0%

COLOPHON

This dissertation uses LaTeXfor the text formatting, BibTeX for the bibliography, and Ti*k*Z for figures and code formatting. LaTeXtemplate for this dissertation adapted from one graciously provided by Will Benton.[1] Edited in the Overleaf cloud editor environment.

[1] https://github.com/willb/wi-thesis-template

BIBLIOGRAPHY

[1] P. Alves, F. Gruber, J. Doerfert, A. Lamprineas, T. Grosser, F. Rastello, and F. M. Q. 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. ISBN 9781450336895. doi: 10.1145/2814270.2814285. URL http://doi.acm.org/10.1145/2814270.2814285.

[2] E. Andreasen and A. Møller. Determinacy in Static Analysis for jQuery. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 17–31, New York, NY, USA, 2014. ACM. ISBN 9781450325851. doi: 10.1145/2660193.2660214. URL http://doi.acm.org/10.1145/2660193.2660214.

[3] 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.

[4] C. Boettiger. An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1):71–79, Jan. 2015. ISSN 0163-5980. doi: 10.1145/2723872.2723882. URL https://doi.org/10.1145/2723872.2723882.

[5] 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. ISBN 9781581136470. doi: 10.1145/780732.780743. URL http://doi.acm.org/10.1145/780732.780743.

[6] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Efficient construction of approximate call graphs for JavaScript IDE services. In D. Notkin, B. H. C. Cheng, and K. Pohl, editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 752–761. IEEE Computer Society, 2013. doi: 10.1109/ICSE.2013.6606621. URL https://doi.org/10.1109/ICSE.2013.6606621.

[7] 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. ISSN 0164-0925. doi: 10.1145/1377492.1377493. URL http://doi.acm.org/10.1145/1377492.1377493.

[8] GNOME. GLib Reference Manual, 1998. URL https://developer.gnome.org/glib/2.46/.

[9] GNOME. The GNOME Project, 1999. URL https://www.gnome.org/.

[10] GNOME. Gck Library Reference Manual, 2003. URL https://developer.gnome.org/gck/3.18/.

[11] GNOME. GIO Reference Manual, 2007. URL https://developer.gnome.org/gio/2.46/.

[12] GNOME. GObjectIntrospection, 2009. URL https://gi.readthedocs.io/en/latest/.

[13] S. Golemon, M. Gusarov, I. The Written Word, E. Fant, D. Stenberg, and S. Josefsson. libssh2, 2009. URL http://www.libssh2.org/.

[14] 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. ISBN 9781595933751. doi: 10.1145/1134285.1134319. URL http://doi.acm.org/10.1145/1134285.1134319.

[15] S. H. Jensen, A. Møller, and P. Thiemann. Type Analysis for JavaScript. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, pages 238–255, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 9783642032363. doi: 10.1007/978-3-642-03237-0_17. URL http://dx.doi.org/10.1007/978-3-642-03237-0_17.

[16] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the Eval That Men Do. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 34–44, New York, NY, USA, 2012. ACM. ISBN 9781450314541. doi: 10.1145/2338965.2336758. URL http://doi.acm.org/10.1145/2338965.2336758.

[17] G. Kastrinis and Y. Smaragdakis. Hybrid Context-sensitivity for Points-to Analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 423–434, New York, NY, USA, 2013. ACM. ISBN 9781450320146. doi: 10.1145/2491956.2462191. URL http://doi.acm.org/10.1145/2491956.2462191.

[18] Y. Ko, H. Lee, J. Dolby, and S. Ryu. Practically Tunable Static Analysis Framework for Large-Scale JavaScript Applications (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 541–551, United States, Nov. 2015. IEEE/ACM. doi: 10.1109/ASE.2015.28.

[19] R. Kohavi, R. Longbotham, D. Sommerfield, and R. M. Henne. Controlled experiments on the web: survey and practical guide. *Data Mining and Knowledge Discovery*, 18 (1):140–181, Feb. 2009. ISSN 1384-5810. doi: 10.1007/s10618-008-0114-1. URL https://doi.org/10.1007/s10618-008-0114-1.

[20] 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. ISBN 978-0-7695-2102-2. URL http://dl.acm.org/citation.cfm?id=977395.977673.

[21] 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. ISBN 9781595935953. doi: 10.1145/1251535.1251546. URL http://doi.acm.org/10.1145/1251535.1251546.

[22] M. Litzkow, M. Livny, and M. Mutka. Condor-a hunter of idle workstations. In *[1988] Proceedings. The 8th International Conference on Distributed*, pages 104–111, June 1988. doi: 10.1109/DCS.1988.12507.

[23] B. Liu, J. Huang, and L. Rauchwerger. Rethinking Incremental and Parallel Pointer Analysis. *ACM Transactions on Programming Languages and Systems*, 41(1):6:1–6:31, Mar. 2019. ISSN 0164-0925. doi: 10.1145/3293606. URL https://doi.org/10.1145/3293606.

[24] 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. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 103–116, New York, NY, USA, 2007. ACM. ISBN 9781595935915. doi: 10.1145/1294261.1294272. URL http://doi.acm.org/10.1145/1294261.1294272.

[25] A. J. Maas, H. Nazaré, and B. Liblit. Array length inference for C library bindings. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 461–471, New York, NY, USA, Aug. 2016. Association for Computing Machinery. ISBN 978-1-4503-3845-5. doi: 10.1145/2970276.2970310. URL https://doi.org/10.1145/2970276.2970310.

[26] L. Mauborgne and X. Rival. Trace Partitioning in Abstract Interpretation Based Static Analyzers. In M. Sagiv, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 5–20, Berlin, Heidelberg, 2005. Springer. ISBN 978-3-540-31987-0. doi: 10.1007/978-3-540-31987-0_2.

[27] 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. ISBN 9781605583921. doi: 10.1145/1542476.1542504. URL http://doi.acm.org/10.1145/1542476.1542504.

[28] H. Nazaré, I. Maffra, W. Santos, L. Barbosa, L. Gonnord, and F. M. Quintão Pereira. Validation of Memory Accesses Through Symbolic Analyses. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 791–809, New York, NY, USA, 2014. ACM. ISBN 9781450325851. doi: 10.1145/2660193.2660205. URL http://doi.acm.org/10.1145/2660193.2660205.

[29] 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. ISSN 0164-0925. doi: 10.1145/1065887.1065892. URL http://doi.acm.org/10.1145/1065887.1065892.

[30] B. B. Nielsen and A. Møller. Value Partitioning: A Lightweight Approach to Relational Static Analysis for JavaScript. In R. Hirschfeld and T. Pape, editors, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:28, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. ISBN 978-3-95977-154-2. doi: 10.4230/LIPIcs.ECOOP.2020.16. URL https://drops.dagstuhl.de/opus/volltexte/2020/13173. ISSN: 1868-8969.

[31] S. Overflow. Stack Overflow Developer Survey 2015. Technical report, Stack Overflow, 2015. URL https://insights.stackoverflow.com/survey/2015/.

[32] S. Overflow. Stack Overflow Developer Survey 2020, 2020. URL https://insights.stackoverflow.com/survey/2020/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2020.

[33] C. Park and S. Ryu. Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity. In J. T. Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 735–756, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 9783939897866. doi: 10.4230/LIPIcs.ECOOP.2015.735. URL http://drops.dagstuhl.de/opus/volltexte/2015/5245.

[34] C. Park, H. Im, and S. Ryu. Precise and Scalable Static Analysis of jQuery Using a Regular Expression Domain. In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS 2016, pages 25–36, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4445-6. doi: 10.1145/2989225.2989228. URL http://doi.acm.org/10.1145/2989225.2989228.

[35] T. Ravitch and B. Liblit. Analyzing Memory Ownership Patterns in C Libraries. In *Proceedings of the 2013 International Symposium on Memory Management*, ISMM '13, pages 97–108, New York, NY, USA, 2013. ACM. ISBN 9781450321006. doi: 10.1145/2464157.2464162. URL http://doi.acm.org/10.1145/2464157.2464162.

[36] T. Ravitch, S. Jackson, E. Aderhold, and B. Liblit. Automatic Generation of Library Bindings Using Static Analysis. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 352–362, New York, NY, USA, 2009. ACM. ISBN 9781605583921. doi: 10.1145/1542476.1542516. URL http://doi.acm.org/10.1145/1542476.1542516.

[37] A. Romano, Y. Zheng, and W. Wang. MinerRay: semantics-aware analysis for ever-evolving cryptojacking detection. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, pages 1129–1140, New York, NY, USA, Dec. 2020. Association for Computing Machinery. ISBN 978-1-4503-6768-4. doi: 10.1145/3324884.3416580. URL https://doi.org/10.1145/3324884.3416580.

[38] 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. ISSN 0164-0925. doi: 10.1145/1057387.1057388. URL http://doi.acm.org/10.1145/1057387.1057388.

[39] J. F. Santos, P. Maksimović, T. Grohens, J. Dolby, and P. Gardner. Symbolic Execution for JavaScript. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*, PPDP '18, pages 1–14, New York, NY, USA, Sept. 2018. Association for Computing Machinery. ISBN 978-1-4503-6441-6. doi: 10.1145/3236950.3236956. URL https://doi.org/10.1145/3236950.3236956.

[40] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A Symbolic Execution Framework for JavaScript. In *2010 IEEE Symposium on Security and Privacy*, pages 513–528, May 2010. doi: 10.1109/SP.2010.38. ISSN: 2375-1207.

[41] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Dynamic Determinacy Analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 165–174, New York, NY, USA, 2013. ACM. ISBN 9781450320146. doi: 10.1145/2491956.2462168. URL http://doi.acm.org/10.1145/2491956.2462168.

[42] Y. Smaragdakis, G. Balatsouras, and G. Kastrinis. Set-based Pre-processing for Points-to Analysis. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 253–270, New York, NY, USA, 2013. ACM. ISBN 9781450323741. doi: 10.1145/2509136.2509524. URL http://doi.acm.org/10.1145/2509136.2509524.

[43] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras. Introspective Analysis: Context-sensitivity, Across the Board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 485–495, New York, NY, USA, 2014. ACM. ISBN 9781450327848. doi: 10.1145/2594291.2594320. URL http://doi.acm.org/10.1145/2594291.2594320.

[44] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation Tracking for Points-To Analysis of JavaScript. In *ECOOP 2012 – Object-Oriented Programming*, pages 435–458. Springer, Berlin, Heidelberg, June 2012. doi: 10.1007/978-3-642-31057-7_20. URL https://link.springer.com/chapter/10.1007/978-3-642-31057-7_20.

[45] The libgit2 contributors. libgit2, 2011. URL https://libgit2.github.com/.

[46] The Telepathy Project. Telepathy-GLib, 2005. URL http://telepathy.freedesktop.org/.

[47] T. S. Ventures. The Two Sigma Ventures Open Source Index, June 2021. URL https://twosigmaventures.com/open-source-index/.

[48] S. Wei and B. G. Ryder. Practical blended taint analysis for JavaScript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 336–346, New York, NY, USA, July 2013. Association for Computing Machinery. ISBN 978-1-4503-2159-4. doi: 10.1145/2483760.2483788. URL https://doi.org/10.1145/2483760.2483788.

[49] S. Wei, O. Tripp, B. G. Ryder, and J. Dolby. Revamping JavaScript static analysis via localization and remediation of root causes of imprecision. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, pages 487–498, Seattle, WA, USA, 2016. ACM Press. ISBN 978-1-4503-4218-6. doi: 10.1145/2950290.2950338. URL http://dl.acm.org/citation.cfm?doid=2950290.2950338.

[50] D. A. Wheeler. SLOCCount, 2004. URL http://www.dwheeler.com/sloccount/.

[51] T. Wies, V. Kuncak, K. Zee, A. Podelski, and M. Rinard. On Verifying Complex Properties using Symbolic Shape Analysis. *arXiv:cs/0609104*, Sept. 2006. URL http://arxiv.org/abs/cs/0609104. arXiv: cs/0609104.