Measuring Mangled Name Ambiguity in Large C/C++ Projects

RICHÁRD SZALAY, Eötvös Loránd University, Faculty of Informatics, Dept. of Programming Languages and Compilers ZOLTÁN PORKOLÁB, Eötvös Loránd University, Faculty of Informatics, Dept. of Programming Languages and Compilers, and Ericsson Ltd.

DANIEL KRUPP, Ericsson Ltd.

Symbol reference has an important role in various areas of software technology. Some modern object-oriented languages use it for implementing function and operator overloading, linkers use it for connecting objects between different translation units, and development environments need to accurately understand them to provide features like traversing all references. In case of the C++ programming language, the most current tools use mangled names to correlate symbols, e.g. when implementing actions like "go to definition" or "list all references". However, for large projects, where multiple binaries are created, symbol resolution based on mangled names can be, and usually is, ambiguous. This leads to inaccurate behaviour even in major development tools. Does this problem inherently stem from the size of the codebase, or is it just another indication of possibly underachieving software quality? In this paper, we discuss the prevalence of this problem on five projects, from open source to industrial code bases with varying degree of code size, to give an input on whether or not the research of a better symbol resolution algorithm is necessary.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors—Compilers; H.1.2 [Models and Principles]: User/Machine Systems—Human Information Processing

Additional Key Words and Phrases: Program comprehension, C++ programming language

1. INTRODUCTION

The size and scale of software systems have grown rapidly over the course of the recent years. Largescale projects amounting up to million lines of code aren't uncommon – e.g. the source code of the Linux kernel, with drivers, is around 17 million lines of code. This presents a challenge when it comes to understanding and navigability of the project. It is always essential to understand the precise behaviour of a software system when we are fixing a bug, or extending the system with a new functionality, and the importance of this understanding is preeminent when a major refactoring task is being undertaken.

To enhance understanding of the project, we have to correlate the occurrences of symbols in the source code and how these symbols are present in the built binaries. This helps developers to understand the project they are working on better via code comprehension tools while helping automated defect detection via static analysis tools. Consider, though, that this correlation usually can't be dis-

Author's addresses:

This work was supported by EFOP-3.6.2-16-2017-00013.

Copyright ©by the paper's authors. Copying permitted only for private and academic purposes.

In: Z. Budimac (ed.): Proceedings of the SQAMIA 2017: 6th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications, Belgrade, Serbia, 11-13.9.2017, Also published online by CEUR Workshop Proceedings (http://ceur-ws.org, ISSN 1613-0073)

Eötvös Loránd University, Faculty of Informatics, Dept. of Programming Languages and Compilers, Pázmány Péter sétány 1/C, Budapest, Hungary, H-1177, szalayrichard@inf.elte.hu, gsd@elte.hu Ericsson Hungary Ltd., Irinyi József utca 4-20., Budapest, Hungary, H-1117, {zoltan.porkolab, daniel.krupp}@ericsson.com

17:2 • R. Szalay, Z. Porkoláb, D. Krupp

covered using simple name identity: in most programming languages, the same name could mean different symbols based on the occurrence's context, e.g. attributes or methods with the same name in different classes, overloaded version of functions, or functions with the same name in different names-paces/packages or build configurations.

Modern software tools, like various development frameworks [Microsoft 2016a; CLion 2016; Net-Beans 2016; Eclipse 2016] and code comprehension tools [Woboq 2016; OpenGrok 2016] provide discovery functionality: the user of the tool can jump to the place of the definition or can iterate all places of reference of a given symbol. Naturally, we expect such tools to work not on simple name identity, but on exact symbol correlation. Unfortunately, we found that current development and comprehension tools fall behind and into ambiguity with the increase of the code base's size. The reason is – as we will explain – that *mangled names* are allowed to be ambiguous when multiple executables are produced to form the released product.

This paper is organised as follows:

We describe the ambiguity problem in Section 2 in details. Current tools' behaviour is evaluated in Section 3. Section 4 presents our measurement results on five projects. The related work with possible future directions are discussed in Section 5. Our paper concludes in Section 6.

2. PROBLEM DESCRIPTION

We used the following criteria on "When is a symbol unambiguously defined?" for the measurements discussed in this paper. We have devised to be sufficient in our C++ context of large projects. In the attached tables, we show our measurements on how many of such problematic symbols were found for our test projects.

A **function symbol** f () (mangled name in LLVM/Clang: $_Z1fv$) is considered ambiguous if neither of the following conditions hold true for all symbols having this particular mangled name in the entire project:

-The function has exactly one *definition* node

-The function does not have a *definition* node, but has exactly one *declaration* node

-The function has neither a *definition* node nor a *declaration* node

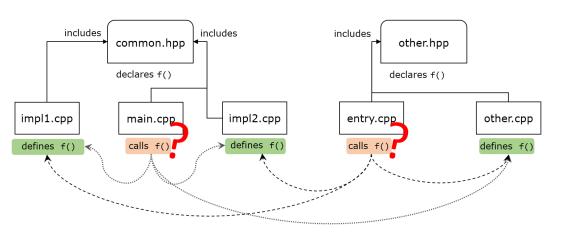
A **type symbol** T (mangled name in LLVM/Clang: T) is considered ambiguous if it is *defined* more than once in the entire project. Types can be *declared* as many times as needed. A header discovered to be included multiple times in the same translation unit does not constitute as multiple definitions, as *header guards* make sure the preprocessed text is well-formed.

In the case of the C++ programming language, usually the *mangled name* [Stroustrup 2013; Ellis and Stroustrup 1990] is used to distinguish between different symbols of the same name. The mangled name is constructed using (possibly multiple) namespace and class information by concatenation, but its exact form is compiler-dependent. Compilers use mangled names to enable *operator overloading* [Stroustrup 1994], i.e. they generate different mangled names for functions with the same name but different parameter lists. Linkers use these mangled names to resolve symbols [ISO C++ 2014a].

In the case of the C language, there is no such thing as namespace, class, or function overloading [ISO C 2011]. Still, in certain cases, such as specific optimisations on how a method is called, *name decoration* occurs [Microsoft 2016b]. In this paper we will refer to the name visible to the linker as *mangled name*, both in the context of C and C++ for the sake of simplicity.

While mangled names must be unique for all translation units linked into a single executable, this does not stand for large-scale projects where multiple executables are typical.

In a software comprehension activity the most frequent questions users ask are "Where is this method called, or type referenced?" and "What does the declaration or definition of this looks like?" [Sil-



Measuring Mangled Name Ambiguity in Large C/C++ Projects • 17:3

Fig. 1. Jumping to the definition of f() from the calls is ambiguous in this example project layout.

lito et al. 2008]. A software comprehension tool should be able to answer these questions as precisely as possible, as accuracy ensures more optimised usage of the developers' time spent working. Both questions lead to the fundamental problem of correctly resolving references to the definition and usages of a type, a function, or variable, and other language components.

According to the *One Definition Rule* (ODR) of C++ [ISO C++ 2014b], only one definition of any variable, function, class type, enumeration type, or template is allowed in any translation unit. When resolving references to ordinary C functions, static and non-virtual C++ member functions, type names or non-polymorphic variables, the unique definition within a single translation unit can be found based on static information. Specifically, the function definition of non-virtual functions or ordinary C functions can be looked up based on function signature, which – according to [ISO C++ 2014c] – contains the name of the function, the enclosing namespace, class of which the function is member of (in the context of C++), the type of the parameters, template parameter list (in case of function templates in C++), *cv*- and *ref*-qualifiers, unique type names (qualified with namespace) and scope-correct variable names.

There can be, however, more than one definition belonging to the same unique-name or signature, but defined in different translation units that are not linked together. This is a typical scenario in large-scale programs consisting of multiple separate executables and build configurations, e.g. every executable having a main() function as entry point.

Since the translation unit containing the reference and the set of translation units linked together is known for the linker, it is possible for the linker to look up the correct, unique definition for any given reference.

In contrast, for a software development or comprehension tool, while the user is browsing a source file, the linkage context (the set of translation units linked together) where the definitions should be resolved is usually unknown [Ferenc et al. 2002]. This leads to ambiguous type, function or variable references. In some cases this ambiguity can be resolved automatically, by taking into consideration the linkage information. For example in Figure 1, the reference f() in entry.cpp can be resolved to the definition in other.cpp, since it is known that entry.cpp is only linked with other.cpp in any target binary.

In some cases the ambiguity can only be resolved by asking the user to specify the linking context, but asking the linking context should only be done when it is absolutely necessary, to avoid superfluous user interaction that causes disorientation in the navigation process [Alwis and Murphy 2006], as

17:4 • R. Szalay, Z. Porkoláb, D. Krupp

developers have to resolve symbol references using their internal and external knowledge instead of a proper tool solving the problem for them.

3. STATE OF THE ART

Current well-known development, engineering, and code comprehension tools rarely provide good support for this problem. We have analysed some common tools in terms of their reaction to a "jump to definition" query:

Microsoft Visual Studio [Microsoft 2016a] shows a disambiguation page when encountering the problem described in this paper. If the entire *solution* (a group of projects handled together) is configured for a certain dependency (i.e. only one of the ambiguous definitions is compiled when the solution is built) and the user changes the internal settings of the solution, Visual Studio decides which symbol a "jump to definition" query jumps to. This fine-tuning on the users' end seemingly does not affect "get calls"/" get usage"-like queries, which still show results with every possible option present, including those which are clearly not valid in the solution's current state.

JetBrains CLion [CLion 2016] analyses and builds symbol information when a project is configured, and one project having multiple separately configured executable targets misleads the IDE. A certain file is designated as location where function f() is defined – our understanding currently reveals that the file which has a larger name in lexicographical order. The ambiguity is present even when a certain executable is being debugged by the IDE with "Step Into" showing the proper implementation being executed while "Jump to Definition" opening an entirely distinct source file.

NetBeans [NetBeans 2016] does not show a disambiguation page at all, seemingly jumping to a file first detected for a certain symbol after the last build – this can be overridden by manually setting certain files to "Exclude from Build" after which the file's symbols won't contribute to the set of potential results. The file to which NetBeans jumps for a definition varies between client restarts, seemingly in a non-deterministic fashion; the only exception is when the symbol is *defined* in the same file where it is *used*: in this case *all* queries jump to this location in particular.

Eclipse [Eclipse 2016] properly prioritises symbols explicitly defined in the same source file, but if the definition is not found in the file where a query is issued (see example in Figure 1), a disambiguation page is shown. Putting different builds into entirely different projects with their own Makefile solves this issue, but search queries do not traverse project boundaries.

Woboq Code Browser [Woboq 2016] shows the locations where a symbol is defined when viewing information about a particular symbol, but the problem of Section 2 is present. Jumping to definition by clicking on a usage location jumps the user to the definition that has been first, in the order of build commands discovered by the codebrowser_generator tool of *Woboq*. It binds usages and definitions in the same source file together.

OpenGrok [OpenGrok 2016] uses Apache Lucene to index source code and it considers source files as pure textual input with no knowledge of build relationships. Thus, it always has to provide a disambiguation page if a mangled name is ambiguously defined.

Software development and comprehension tools usually, at most, show a disambiguation page and put weight on the developer's shoulders to make out which is the "real" definition said call is referring to.

4. MEASUREMENTS AND RESULTS

We used LLVM/Clang to extract information from the Abstract Syntax Tree (AST) generated by parsing our analysed projects. This information has been saved into a database, on which we ran our investigations.

	Functions					
	Names	Nodes	Decl.	Def.	U sage	
Xerces [Apache 2016]	18547	63167	14933	11123	37111	
	(19, 0.1%)	(348, 0.5%)	(23%)	(18%)	(59%)	
CodeCompass [CodeCompass 2016]	337833	622220	229737	127864	264619	
	(143, 0.04%)	(473, 0.07%)	(37%)	(20%)	(43%)	
LLVM [Clang 2016]	1793978	4545086	1213123	708156	2623807	
	(334, 0.02%)	(16469,0.4%)	(27%)	(16%)	(57%)	
Linux [LLVM/Linux 2016]	70579	411691	33746	66228	311717	
	(503, 0.7%)	(11074 (2.7%)	(8%)	(16%)	(76%)	
TSP [Ferraro-Esparza et al. 2002]	410946	2091578	377393	321964	1392221	
	(9721, 2.3%)	(1941189.3%)	(18%)	(15%)	(67%)	

Table I. Distribution of ambiguity and symbol types amongst functions

We have measured the problem on different projects, such as Apache Xerces [Apache 2016], Code-Compass [CodeCompass 2016], LLVM/Clang infrastructure [Clang 2016], LLVM/Linux [LLVM/Linux 2016] (the GNU/Linux operating system ported to be parseable and compilable on LLVM/Clang) and an Ericsson platform product, TSP [Ferraro-Esparza et al. 2002].

Attached tables provide an overview on the quantities. To outline the occurrence of the problem, we must investigate the number of distinct symbol **names**, i.e. how many mangled-name equivalence groups can be found in the project. The number of **nodes** show how many times a particular symbol (function or type) appears in the source code. Each node has a name, the symbol it references. The total number of *declarations*, *definitions* and *usages* (function calls, the name of a type appearing in a function signature or a variable declaration) is also presented.

In theory, the number of definitions should equal the number of unique names (assuming that the project is a single binary, because, as *ODR* states each function, global variable, etc. must have exactly *one* definition), but this is clearly not the case: first of all, software projects aren't comprised of a single binary, and some symbols are resolved dynamically.

To understand the impact of the problem on developer effort and navigation costs, we have to visualise the individual groups and see the ambiguity in detail. For example, in Xerces, 19 unique function names are considered ambiguous, and these names are used in 348 locations. The *distribution* of this ambiguity is an important factor of impact, as each non-unique name having 18 nodes attached (348/19 = 18, so the ambiguity is averaged out) is not as powerful, as 1 important name (one that is frequently navigated by developers, for call sites and definitions) having ~ 300 usages would be. Figure 2. shows the distribution of the number of definition symbols for a mangled name group.

We have seen that while the problem appears to affect a considerable ratio of symbols (0.5 - 10%) if the project is viewed as a whole, the individual cases of ambiguity is resolvable by the developers easily. On the other hand, automated toolchains, such as static analysers, can fall into indecision when meeting this ambiguity, as better heuristics could help symbol resolution.

In the following subsections we will discuss in detail the findings in every project we have evaluated individually.

4.1 Xerces

Apache Xerces [Apache 2016] is an open-source XML parser library framework. The most prominent ambiguity is from the StrX class and its two member functions, which are defined in the *tests* and *samples* folder. They account for 55% (4 names having 15 definition each, and 182 call sites in total) of the duplication. The usage() function, existing only in these sample projects, accounts for the remainder of the duplication in Xerces, totalling 16 definitions with 37 call sites, which are individually located in their respective *sample* projects.

17:6 • R. Szalay, Z. Porkoláb, D. Krupp

	Types				
	Names	Nodes	Decl.	Def.	U sage
Xerces [Apache 2016]	2849	60715	2248	1402	57065
	(9, 0.3%)	(329, 0.5%)	(4%)	(2%)	(94%)
CodeCompass [CodeCompass 2016]	346387	779523	182413	169124	427986
	$(1315\ (0.4\%)$	(9461, 1.2%)	(23%)	(22%)	(55%)
LLVM [Clang 2016]	1628232	4206743	860245	789886	2556612
	(7192, 0.5%)	(73375, 1.8%)	(20%)	(19%)	(61%)
Linux [LLVM/Linux 2016]	19883	159581	1990	20029	137562
	(188, 1%)	(579, 0.3%)	(1%)	(13%)	(86%)
TSP [Ferraro-Esparza et al. 2002]	196374	2108217	62914	162226	1883077
	(2478, 1.2%)	(74511, 3.5%)	(3%)	(8%)	(89%)

Table II. Distribution of ambiguity and symbol types amongst types

Thus, in Xerces, a developer working on the "actual" project is not likely to be led off-road by the ambiguity, and in the test/sample projects, the only decision the developers need to take in case of a disambiguation page, is to select the referenced symbol within the same test/sample. However, this deeper inspection of the code revealed a smell: the implementation of StrX is duplicated in all the test projects – which could make changes to the test/sample infrastructure a hard task.

4.2 LLVM/Clang

In case of LLVM ¹ [Clang 2016], certain functions, such as fieldFromInstruction and getMnenomic, etc., related to different targeted architectures form the main mass of duplication. In case of the first function, we measured 6244 call sites to 7 definitions. Only 650 of the calls were not in the same source file as a definition, however, in this case, a disambiguation is easily resolvable by using the source file which corresponds to the compilation architecture the developer is currently working on or navigating.

A problem can, however, arise for static analysis tools or any automated tool which would expect mangled names to be unique on the project level, as in this case, the tool needs to be armed with a heuristic that can figure out which implementation to use – due to LLVM's high-grade code quality, a simple "similar looking source code path" rule could be enough.

4.3 LLVM/Linux

The LLVM/Clang parser compatible version of the GNU/Linux kernel, called LLVM/Linux [LLVM/Linux 2016] (the original source code cannot be parsed by Clang due to usage of non-standard, GNU GCC specific extensions) contains type duplication which are almost all only a 2 definition kind. We need to point out that it is C, and not C++ code, and Linux is an operating system's kernel, so the language rules and the developer intentions are different in this case. However, these duplicate definitions are entirely separate in their use and Application Binary Interface (ABI). Examples include module and file. Based on the name, it is not easily seen which instance should be used in a given context, as some member fields overlap.

Almost all ambiguous function definitions fall into the 2 definition category too. These result from the fact of straight code duplication, to which most headers and source files state Copied from <PATH> in their preamble. Either in case of fully copied files or copied bits of code, we did not find *different* implementation – in terms of source code text, and not program semantics! However, a modification to

 $^{^{1}}LLVM$ is an open source project backed by software companies worldwide providing a unified way of generating machine code from the LLVM Internal Representation (IR) language. *Clang* is a frontend suite, which translates C/C++ source code to LLVM IR. The two projects together make up for a versatile C/C++ compiler.

either of the ambiguous definitions most likely will require a change to the other, which could bring forth undocumented or unforeseen consequences to the kernel as a whole.

The Linux kernel has been debated for its code quality since it started gaining popularity. The above examples show, that a "happens-to-work" mentality stands as an obstacle for current and future developers, which can hinder the evolution of the project. Easy understanding, security and safety in an operating system should be a must, and we can just again talk about static analysis tools, which are majorly suffering due to ambiguity.

4.4 TSP

Ericsson's Telecom Server Platform [Ferraro-Esparza et al. 2002] combines multiple well-known open source and industrial, proprietary technologies. Because of this, most of the Linux kernel's issues are applicable for TSP too. Due to its nature of embedded and enterprise-grade usage, TSP combines the use of many different C and C++ standard library implementations, along with custom memory management and a modular interface. All these contribute to duplicate definitions for functions and types having general names, such as setup_module, teardown or Transaction. Most of these references can be resolved if the developer, or tool, selects the appropriate definition based on a "file paths look alike" heuristic.

In case of the different standard libraries and memory managers, it is not apparent which implementations are used. The developer must have the knowledge beforehand, or try every possible implementation: some of these program elements have ≥ 6 different definition bodies to the same symbol name.

5. RELATED AND FUTURE WORK

Producing clusters by decomposing large-scale software. Richard C. Holt emphasises on the importance of linkage information when it comes to extracting software models in [Wu and Holt 2004]. Their approach uses the information retrieved from the linker to resolve dangling references in software models created by extractors, such as [Ferenc et al. 2001].

We would like to increase our data by investigating – perhaps with a computer-aided solution – more projects. The previous heuristics mentioned (such as "file paths look alike") is a topic for further research, to view how powerful such an approach would be in aiding developers and static analysis tools.

Further heuristics should be discussed and developed that could further reduce the list of ambiguous symbols, e.g. parts of the Standard Template Library could be excluded from the ambiguity check process via recording the actual version of the STL used when building the project.

A problem similar to that described can be applied to other programming languages, e.g. in *Java*, different classpaths could be considered as "linkage" information and the question of "How symbols reference each other?" is applicable for packages built with different configuration in a large-scale project.

6. CONCLUSION

In this paper we investigated the ambiguity of symbol resolution based on mangled names in large C/C++ programs. Resolving symbol references is the base of many essential functionalities – such as navigating to the definition of a type or function, listing all call sites of a given function – implemented in software development tools. Most of the available development environments and comprehension tools implement these features for C/C++ based on mangled names. Unfortunately, using mangled names only may lead to ambiguity when different symbols with the same mangled name are defined in different translation units and assembled into different binaries.

17:8 • R. Szalay, Z. Porkoláb, D. Krupp

We have shown that while current state of the art tools are ignoring this problem, and on a project level, a considerable ratio (0.5 - 10%) of symbols are affected, the problem can easily be resolved by human developers. On the other hand, automated tools, such as static analysers, could be in need of heuristics, such as source path similarity, to enhance their symbol resolution across translation units.

REFERENCES

[Microsoft 2016a] Microsoft Visual Studio, version 14.0.24720.00-Update1, https://www.visualstudio.com/, 08/15/2016.

[CLion 2016] CLion: a cross-platform IDE for C and C++, version 2016.2.1, https://www.jetbrains.com/clion/, 08/15/2016.

[NetBeans 2016] The NetBeans IDE, version 8.1, https://netbeans.org/, 08/15/2016.

[Eclipse 2016] The Eclipse project, Eclipse CDT (C++ Development Tools), version Neon 4.6.0, http://www.eclipse.org/home/, 08/15/2016.

[Woboq 2016] Woboq code browser, version 2.0.1, https://woboq.com/codebrowser.html, 08/15/2016.

[OpenGrok 2016] OpenGrok home page, version 0.13-rc2, https://opengrok.github.io/OpenGrok, 08/15/2016.

[Stroustrup 2016] Bjarne Stroustrup: The C++ Programming Language, 4th ed., Addison-Wesley, ISBN 978-0321563842, 2013. [Ellis and Stroustrup 1990] Margaret A. Ellis and Bjarne Stroustrup: The Annotated C++ Reference Manual. Section 7.2.1c.

ISBN 0-201-51459-1, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 1990.

[Stroustrup 1994] Bjarne Stroustrup: Design and Evolution of C++, Addison-Wesley, ISBN 0-201-54330-3, 1994.

[ISO C++ 2014a] The C++14 Standard, ISO International Standard,

ISO/IEC 14882:2014 — Programming Language C++, §1.3 "Terms and definitions", 2014.

[ISO C 2011] The C11 Standard, ISO International Standard, ISO/IEC 9899:2011 — Programming Language C, 2011.

[Microsoft 2016b] Format of a C decorated name, https://msdn.microsoft.com/en-us/library/x7kb4e2f.aspx, 08/15/2016.

[Sillito et al. 2008] Jonathan Sillito, Gail C. Murphy, Kris De Volder: Asking and Answering Questions during a Programming Change Task, IEEE Transactions on Software Engineering, VOL. 34, NO. 4, July/August 2008.

[Ferenc et al. 2002] Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy: Columbus Reverse Engineering Tool and Schema for C++, In Proc. 18th Int'l Conf. Software Maintenance (ICSM 2002), pp. 172–181., Oct. 2002.

[Alwis and Murphy 2006] B. De Alwis and G.C. Murphy: Using Visual Momentum to Explain Disorientation in the Eclipse IDE, Proc. IEEE Symp. Visual Languages and Human Centric Computing, pp. 51–54., 2006.

[Apache 2016] Apache Xerces C++, part of Apache Xerces Project, git version 26f8004, http://xerces.apache.org, 08/15/2016.

[CodeCompass 2016] The CodeCompass project, https://github.com/Ericsson/CodeCompass, 08/15/2016.

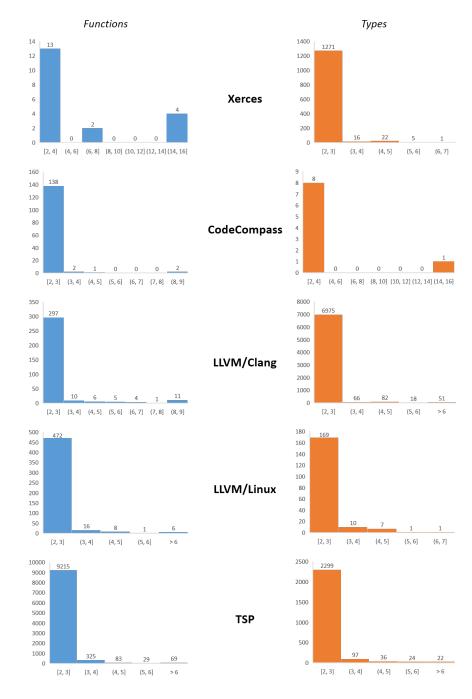
[Clang 2016] Clang: a C language family frontend for LLVM, http://clang.llvm.org, 08/15/2016.

[LLVM/Linux 2016] LLVM/Linux: GNU/Linux kernel compatible for LLVM/Clang builds, git version 230b22d, http://llvm.linuxfoundation.org, 08/15/2016

[Ferraro-Esparza et al. 2002] Ferraro-Esparza, Victor, Michael Gudmandsen, and Kristofer Olsson: Ericsson Telecom Server Platform 4, Ericsson Review 3 (2002) pp. 104–113., accessed 08/15/2016.

[Wu and Hold 2004] Jingwei Wu and Richard C. Holt: *Resolving Linkage Anomalies in Extracted Software System Models*, Proceedings of the 12th IEEE International Workshop on Program Comprehension, pp. 241–, 2004.

[Ferenc et al. 2001] Rudolf Ferenc, Árpád Beszédes, Ferenc Magyar, and Tibor Gyimóthy: A short introduction to Columbus/CAN, Technical Report, 2001.



Appendix

Fig. 2. Distribution of *definition* ambiguity amongst symbols that are ambiguously defined. The numbers shown are the **node** counts.