Source versus Object Code Extraction For Recovering Software Architecture

Ahmed E. Hassan, Zhen Ming Jiang, and Richard C. Holt
Software Architecture Group (SWAG)
School of Computer Science
University of Waterloo
Waterloo, Canada
{aeehassa, zmjiang, holt}@uwaterloo.ca

ABSTRACT
The architecture of many large software systems is rarely documented and if documented it is usually out of date. To support developers maintaining and evolving these systems, an up to date view of the architecture could be recovered from the system's implementation. Source code or object code extractors may be used to recover the architecture.

In this paper, we explore using two types of extractors (source code and object code extractors) to recover the architecture of several large open source systems. We then investigate the differences between the results produced by these extractors to gain a better understanding of the benefits and limitations of each type of extractor. Our experimental results show that both types of extractors have their benefits and limitations. For example, an object code extractor is easier to implement while a source code extractor recovers more dependencies that exist in the source code as seen by developers.

1 INTRODUCTION

Software architecture documents show the main subsystems of a software system and the interaction between these subsystems. For example, the architecture of an operating system may indicate that it has the following subsystems: File System, Memory Manager, Network Interface, Process Scheduler, and Inter-Process Communication subsystems. The architecture may also show that the Memory Manager subsystem depends on ($i.e.$ uses functions or data structures defined in) the File System subsystem in order to swap pages to disk.

All too often the architecture of a software system is not well-documented or is not up to date. Software developers working on large software systems can analyze the implementation of a software system to recover its architecture [3, 6]. An extractor analyzes the implementation of the software system and produces a dependency graph. The dependency graph shows how software entities such as variables, macros and functions depend on each other. An extractor produces tuples such as "`calls funcA funcB`" or "`calls fileA.c fileB.c`". The software architecture is derived by lifting extracted low level dependencies between functions or files (such as `funcA` or `fileA.c`) to high level dependencies between subsystems (such as the Memory Manager subsystem).

Several types of extractors can be used to recover the architecture. For example, an extractor could recover the dependency graph from the source code or the object code. The executable could be analyzed at runtime as well to produce a dynamic dependency graph. For programming languages which have macros, such as C and C++, either the source code or the preprocessed code could be analyzed. In short, the architecture of a software system can be recovered by analyzing the source code or any of the intermediate code representation throughout the build process. Figure 1 gives an overview of the build process for a C application using GNU build tool (*gcc*):

- The Preprocessor (*cpp*) examines the source code files, removes all comments, and expands include files, macro definitions and all other preprocessor directives.
- The Compiler (*cc1plus*) takes the preprocessed source code and produces object code.
- The Linker (*ld*) combines the object files along with necessary system libraries to produce one single executable program.

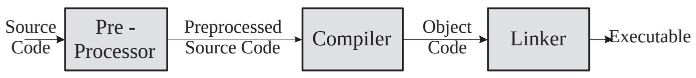

Figure 1: The Build Process For a C Software Application

Recovering the architecture by analyzing the various kinds of intermediate code has its benefits and limitations. For instance, an object code extractor is relatively easy to implement since it requires analyzing object files which have a simple format that is not as complex as analyzing the source code. Unfortunately, an object code extractor produces dependencies based on the source code after the expansion of macros and compiler optimizations. These dependencies are likely to confuse software developers who are more comfortable dealing with source code with macros in it instead of the code after the expansion of macros. On the other hand an extractor which analyzes the source code before macro expansion will produce a more familiar dependency graph to a developer who works at the source code level but on the other hand such an extractor is difficult to implement due to the complexity associated with both macro expansion and source code syntax. Furthermore, the dependency graph produced by an object code extractor represents a single possible configuration of a software system, whereas a source code extractor would produce a more complete dependency graph which corresponds to the various build configurations of the software system.

In this paper, we investigate the dependency graphs produced by two types of extractors (source and object code extractors). We analyze several large open source software systems using both types of extractors. We then investigate the differences between the dependency graphs produced by these extractors. The focus of our comparison is on determining which differences are due to the extraction technique used and which are due to limitation or bugs in the implementation of an extractor. In particular, we seek to understand the following issues:

- The differences between the dependency graph produced by a source code extractor and an object code extractor.
- The amount of missing dependencies in a source code extractor which are due to the complexity of developing such an extractor since implementors of such an extractor would need to deal with legacy code that may be hard to analyze.

Prior related work which studied dependency extractors has focused on evaluating extractors by:

- Comparing extractions on a small manually crafted software systems with constructs known to be hard to extract [1].
- Comparing extractors on different software systems [7].
- Investigating the different types of reports (call trees, data and control graphs) produced using the extracted data [2].

In contrast, the work presented in this paper focuses on performing our analysis on several larger open source case studies. We investigate the differences carefully in a semi automated fashion to understand better the reasons behind the differences (differences due to bugs versus others due to the extraction technique), instead of simply pointing out the differences between the extracted dependency graphs.

Organization of the Paper
The organization of the paper is as follows. In Section 2, we compare the features and performance of the extractors used in our study. In Section 3 we explain the process we used to compare the facts produced by both extractors. In Section 4 we present the results of our comparative empirical experiment. Section 5 discusses lessons that we learned from our experiment. Section 6 concludes the paper.

2 OVERVIEW OF THE TWO EXTRACTORS USED IN OUR ANALYSIS: *LDX* AND *CTAGX*

In this paper, we compare the extracted facts for large software systems at the source code level using a source code extractor called *CTAGX* and at the binary object file level using an object code extractor called *LDX*. We now give an overview of both extractors. We present the extraction technique used by them and the type of data produced by them.

LDX: A Linker Based Extractor
LDX is a linker-based fact extractor which operates on the object code [9]. It is based on a customized version of the GNU linker *ld*. The *LDX* extractor makes use of the linker's knowledge of function and module dependencies to produce program information such as a call graph and variable usage.

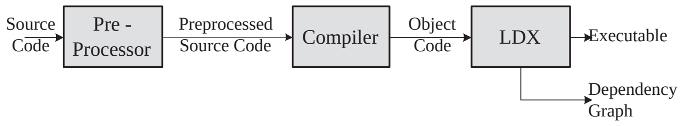

Figure 2: The *LDX* Extraction Process

Figure 2 shows the *LDX* extraction process. *LDX* integrates into the build process of a software application and produces the executable along with the dependency graph for the analyzed software system.

The schema for the data produced by *LDX* is shown in Figure 3. The schema has three types of relations: Kind, Define, and Use.

- **Kind:** identifies the entity type. An entity is either of type "cFunction" or "cKind". "cFunction" indicates that the entity is a function; whereas an entity of type"cKind" can be of type global variable. *LDX* is limited to the type of information that is available at the linking phase, for example, the schema does not contain information about macros or comments in the source code since the preprocessor would have already removed the comments and expanded the macros in the code analyzed by *LDX* as shown in Figure 2.

- **Define:** provides location information for the object file where a particular entity is defined. For example, "`define entityA fileC.o`" indicates that "entityA" is defined in the object file "fileC.o".

- **Use:** describes the dependency relations between entities. For example, "`use entityA entityB`" means entitiyA calls a function called "entityB", or uses a global variable called "entityB".

CTAGX: A *ctags* Based Extractor
Unlike *LDX* which operates on the object code, *CTAGX* operates on the source code. It is based on a source code tagging tool called *ctags* [4] which recovers information about the software system. By adopting *ctags*, *CTAGX* does not need to consider a large number of peculiarities of legacy

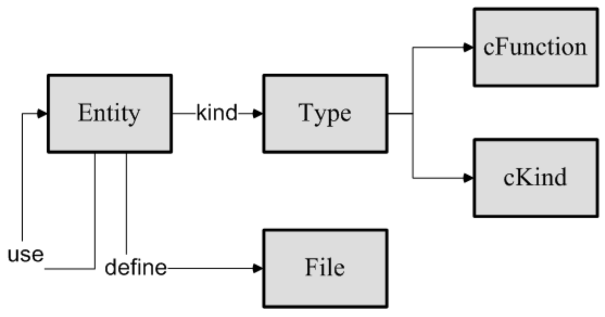

Figure 3: Schema for the Facts Produced by *LDX*

source code. For example, it does not need to worry about ANSI and K&R C differences in the source code. Moreover, *ctags* uses several heuristics to robustly parse source code containing `#if` preprocessor conditional constructs. *ctags* also uses a conditional path selection heuristic to resolve complicated choices. It employs a fall-back strategy when all heuristics fail. Finally, *ctags* has been highly optimized and can parse and tag source code very quickly and efficiently.

CTAGX is not integrated into the build process, instead it analyzes all the .h and .c files in the source code directory of a software system. To extract facts from these files:

- *CTAGX* invokes *ctags* for each file in the software system to identify the type of every defined entity in the file along with the entity's beginning line number. This information for all files is combined to build a *Global Symbol Table*.
- The entity beginning line number is used by *CTAGX* to retrieve the non-commented code content of each defined entity. Tokens in the code content of each defined entity are checked against the *Global Symbol Table* to produce the dependency graph: If a token exists in the symbol table and it exists in the code content of a particular entity, then *CTAGX* creates a dependency between that entity and the entity corresponding to the token found in the symbol table. The content of each entity is used as well to retrieve additional entity information such as the comment tokens for each entity, its parameters, its return type, and a listing of the control keywords used in an entity. To deal with entities with similar names, *CTAGX* combines the contents of entities with similar name into the content of a single entity with a common name.

Comparison of the Two Tools

Since *LDX* analyzes the binary object code, it more likely to produce more accurate results than *CTAGX*. The preprocessor has already removed the comments, expanded the includes, macros and other directives in the code analyzed by *LDX*, therefore the code represents a picture which is much closer to the program's runtime behavior.

However, an object code extractor, like *LDX*, has a number of drawbacks:

- It cannot process code that has syntax errors or code that could not compile. Uncompilable code can be caused by missing build configurations or missing included files, etc. The ability to handle incomplete and incompilable code is crucial during software evolution studies since such studies require the analysis of every version of the the source code and the source code may not compile for several versions.
- The generated dependencies do not match the expectations of a developer who is accustomed to viewing the source with all macros, includes and other directives unexpanded. Moreover an object code extractor is unable to produce detailed information below the function level.

Property	*LDX*	*CTAGX*	*CPPX*
Extracted Facts Level	Function	Statement	Expression
Ease Of Development	Easiest	Medium	Hardest
Analysis Technique	Exact	Heuristic	Exact
Development Technique	Extend	Post-Process	Extend
Supported Languages	C, C++	C	C, C++
Handles Incomplete Code	N	Y	N
Handles Incompilable Code	N	Y	N
Extracts Comments	N	Y	N
Extracts Macros	N	Y	N
Build Time View	Y	N	Y

Table 1: Comparison of Properties of *LDX*, *CTAGX*, and *CPPX*

A source code extractor like *CTAGX* has its advantages compared to *LDX*:

- *CTAGX* produces a view which is closer to the dependency structure which developers would expect to see based on their interaction with the code as shown in an editor, since *CTAGX* produces facts in which all comments are preserved; all macros are unexpanded; and all `#ifdef` conditional compilation branches are considered.
- *CTAGX* is quite robust, since it can analyze uncompilable code by recovering facts from a file while skipping the sections of the code which it fails to parse.
- *CTAGX* can supply information below the function level, such as function return types, and parameter types, local variable usage, etc.

CTAGX has its limitations as well. It processes all the .c and .h files in the source code directory, but it does not analyze Makefiles or attempt to preprocess the code. Thus the facts produced by *CTAGX* are likely to be less accurate in respect to a software's run-time behavior, since *CTAGX* does not consider issues that arise during the build process such as compilation flags.

Table 1 compares the properties of *CTAGX* and *LDX*. The table as well compares both extractors to the *CPPX* extrac-

system	build (*LDX*)	ctags	*CTAGX*
Openssh-3.9p1	47.339s	2.00s	19.9s
Postgres-7.3.4	2m 27.257s	9.19s	37.22s
Linux-2.6.1	25m 1.485s	1m 58.69s	21m 33s

Table 2: Time Statistics for Running Both Extractors on Openssh, Postgres, and Linux

tor which analyzes preprocessed source code [5]. None of the extractors is implemented from scratch, instead they each make use of other open source tools. *CTAGX* post processes *ctags*' output. *LDX* and *CPPX* are implemented by modifying the source code of the *ld* linker and the *cc1plus* compiler respectively. *CPPX* performs more detailed analysis to the source code and can produce data at the expression level but it does not have access to comments or macros. Since the *CPPX* extractor is based on a regular compiler it cannot deal with incomplete (such as missing header files or build configurations) or uncompilable code (such as code with syntax errors). Similarly *LDX* needs the source code to compile so it can extract information from the produced object code. *CTAGX* uses several heuristics to recover gracefully from incorrect code and to analyze `#ifdef` branches, in contrast the other two extractors that use well defined grammars.

All three extractors have been developed by members of our research group. The need to modify the source code of the compiler and the depth of the extracted facts (expression level) makes *CPPX* the hardest to implement. *LDX* is the simplest to implement since the binary file syntax is straight forward in comparison to the grammar of modern programming languages such as C or C++. In this paper, we focus our analysis on the *CTAGX* and *LDX* extractors which are examples of source and object code extractors.

The analysis time needed to recover a dependency graph varies from one system to the other. The time needed for either the (*LDX* or *CTAGX*) extraction technique is fast in comparison to the build time of the analyzed software application. *LDX* requires a few additional seconds beyond the build time of an application. *CPPX* requires almost twice as much time as needed to build an application. *CTAGX* is much faster than *LDX* and *CPPX*. The time needed for *CTAGX* compromises of the time needed to run *ctags* as well as the time needed to analyze the *ctags* output. Table 2 shows the time statistics for extracting the three open source software systems used in our study: Openssh version 3.9p1, Postgres version 7.3.4, and Linux version 2.6.1. For example for Postgres version 7.3.4, it takes about 2 minutes to build and extract facts using *LDX*, it takes about 9 seconds to run *ctags* on the system, and it takes about 37 seconds to extract facts using *CTAGX*. Note that the Linux build is timed using the full configuration setting where all possible build configuration options are enabled.

In summary, both extractors can recover a dependency graph for large software systems in a reasonable time. Yet, each extractor has its limitations due to the used extraction technique.

3 COMPARISON PROCESS

In this section, we present the comparison process we used to study the difference between both extraction techniques. Figure 4 gives an overview of the process. Facts are recovered from a software system using both extractors. The *CTAGX* facts are reduced to only reflect the type of facts that are available in *LDX* facts, in particular facts relating to comments and to information inside of a function such as the number of conditional statements are removed. The modified facts for both extractors record only that an entity exists and track which other entities it depends on.

Using a simple *Perl* script, we compute the difference between the facts produced by each extractor. Using a set of *Perl* and *grep* scripts, we semi-automatically go through the differences to determine if the differences are due to bugs or implementation decisions in the extractor, or if they are due to the used extraction technique. The scripts automate the difference analysis process by locating the reported locations of entities and showing us code snippets surrounding the entity. Using the output of the scripts, we are usually able to determine if a particular difference is due to a bug, an implementation decision, or a limitation of the used extraction technique. In some cases, we had to closely monitor the build process by recording the build session and studying the output of the preprocessor for a few files which had troublesome entities[1]. If we determined that a difference was due to a bug in the implementation, we updated the implementation of the extractor and re-extracted the facts and repeated our comparison process.

Our comparison process has permitted us to not only determine accurately the differences between both extraction techniques but to improve as well the implementation of both extractors. Throughout our analysis we discovered and addressed several issues, for example:

- For *LDX*, we added an option to prevent the output of system calls since they are not outputted by *CTAGX*.
- We updated the *LDX* implementation to deal correctly with static variables defined inside functions. In the example below, *LDX* would output a dependency between "`func`" and "`static_var.0`". Such a dependency existed in the object code but it is not relevant when studying dependencies between source code entities since "`static_var`" is not accessible outside "`func`" and *LDX* should be outputting relations only at the function level. For the studied systems, *LDX* had extracted several static variables defined inside a func-

[1] We studied the output of the preprocessor by recording the build session then re-running the compiler using the same command line parameters as recorded in the session. We also added the "-E" flag to cause the *gcc* compiler to stop after the preprocessing phase and output the preprocessed source code.

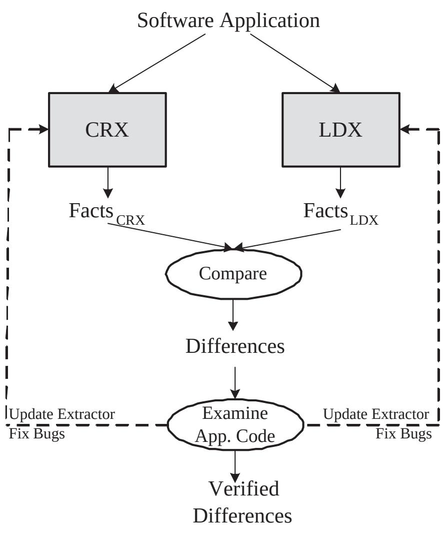

Figure 4: An Overview of our Comparison Process

tion since they are recorded by the linker in the object file. The following code snippet shows an example of a static variable defined inside a function:

```
void func() {
  static int static_var = 1;
}
```

- We fixed a number of bugs that were discovered in *LDX*.
- For *CTAGX*, we fixed several bugs. In particular, a number of heuristics used by *CTAGX* to determine the content of a source code entity were updated to address some of the issues that were discovered during the comparison of the CTAGX and LDX facts.
- We updated *CTAGX* to correctly process C++ style comments ("//") in C code since the studied Linux system made use of such comment style which is supported by the GNU gcc compiler.

Wherever possible we fixed the bugs we found in the implementation of the used extractors. For *CTAGX* we fixed the bugs ourselves. For *LDX* we contacted its developer who fixed the bugs. In some cases, we could not easily fix the bugs in the implementation instead we just left the bugs and noted them in our analysis as a limitation of the tool but not a limitation of technique itself. For example, we discovered two bugs in the CTAGX extractor which are partly due to the complexity associated with parsing source code before the expansion of macros:

- **Complex Conditional Directives:** We discovered that *ctags*, which is used as the backend of *CTAGX*, fails to identify entities if there are too many levels of nested conditional directives. This problem mainly occurs in yacc generated source code files. For example, *ctags* fails to locate entities such as "plpgsql_yyparse" in "src/pl/plpgsql/src/pl_gram.c" and "yyparse" in "src/backend/parser/gram.c". The following code fragment from "src/backend/parser/gram.c" in Postgres is an example of this case:

```
#ifdef YYPARSE_PARAM
# if defined(__STDC__) || defined(__cplusplus)
int yyparse (void *YYPARSE_PARAM)
# else
int yyparse (YYPARSE_PARAM)
  void *YYPARSE_PARAM;
# endif
#else /* ! YYPARSE_PARAM */
#if defined (__STDC__) || defined (__cplusplus)
int
yyparse (void)
#else
int
yyparse ()

#endif
#endif
{ /* function body ... */ }
```

- **Confusion in Determining the Name of an Entity:** *ctags* is not able to correctly determine the name of a variable if a macro or a compiler directive is used on the same line. The following code fragment from "linux-2.6.1/arch/i386/kernel/setup.c" in Linux is an example of this case:

```
int disable_pse __initdata = 0;
```

  ctags mistakenly assumes that the last token before an equal sign (=) is the name of a defined variable. However, based on comments in "linux-2.6.1/include/linux/init.h", the `__initdata` is used to flag initialized data to the Linux kernel.

4 COMPARATIVE EXPERIMENT

We now detail the analysis we performed to compare the output of the two extractors. We conducted our comparative experiments using three open source software systems:

1. **Openssh** which implement a secure SSH network protocol. It includes the ssh, scp, and sftp programs. For our study we used version 3.9p1 which has about 72,000 lines of code.
2. **Postgres** which is a free SQL compliant object Relational Database Management System (DBMS) that originated at the University of California at Berkeley in 1996. For our study we used version 7.3.4 which has about 480,000 lines of code.
3. **Linux** which is an operating system that runs on several platforms and is developed by a large number of developers worldwide. For our study we used Linux version 2.6.1 which has about 4,900,000 lines of code.

After applying both *LDX* and *CTAGX* to the studied systems, we started our comparison process, verifying the differences, and updating the implementation of an extractor if we found a bug. Our comparison process (the center oval in Figure 4) consists of two phases: comparison of entities and comparison of dependencies.

Comparison of Entities
Since *CTAGX* analyzes all the `#ifdef` paths and all the files in the software system, we expected that the entities defined in the *CTAGX* facts would be a super set of the entities defined in the *LDX* facts. However, our experiment shows that this is not the case instead the relation is more like the right hand side of Figure 5: *CTAGX* has more entities defined than *LDX*, but there are certain entities detected in *LDX* that were not detected by *CTAGX*.

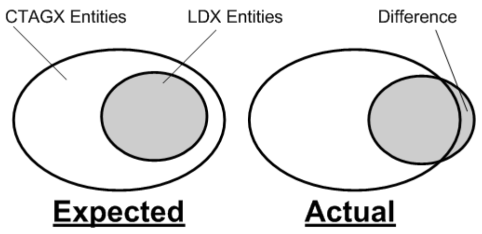

Figure 5: Expected and Actual Entity Comparison Results

Using the comparison process outlined in Figure 4 and through manual inspection of the source code for several differing entities, we categorized the differences into four main categories:

1. Differences due to *LDX* incorrect output.
2. Differences due to Build Process.
3. Differences due to Preprocessor.
4. Differences due to *ctags* limitations.

Table 3 summarizes the results of our comparison process for each software system. For example, the *LDX* output Openssh has 1,529 entities, 88 of these entities do not exist in the *CTAGX* output. 78 of these 88 entities are due to incorrect handling of static variables inside functions by *LDX* and 10 entities are due to preprocessor peculiarities. In the following subsections, we discuss the difference due to the Build process and the Preprocessor. The differences due to *LDX*' incorrect output and *ctags*' limitations were already explored in the previous section.

Differences Due to Build Process
By analyzing the build process for the studied application, we uncovered that in many cases a large number of files are created during the build process. Since *CTAGX* analyzes the source code without taking into account the build process, its facts do not contain any information about these generated files. For example, the *LDX* facts have that the "fmgr_nbuiltins" and "fmgr_builtin" entities are defined in "postgresql-7.3.4/src/backend/utils/fmgrtab.o". Our analysis of a build session of Postgres reveals that these entities are defined in the "fmgrtab.h" file which is created during the building of Postgres.

In addition to automatically generated files, we discovered that in Postgres a few source code files have the suffix "`.map`" instead of having commonly used suffixes such as "`.h`" and "`.c`". For Postgres, we repeated our *CTAGX* extraction and added "`.map`" files to the list of files that *CTAGX* analyzes to make sure the entities in these files are extracted.

Differences Due to Preprocessor
We uncovered two reasons to explain the difference for the remaining entities between *CTAGX* and *LDX*. These two reasons are due to the use of the C Preprocessor to:

- **Declare an Entity**:
  Especially in Linux, there are many entities declared by macros. For example, the data structures "class _device_attr_crypt" and "class _device_attr_beacon" in "linux/net/core/net-sysfs.c" are declared by a macro called "WIRELESS_SHOW" using the following syntax:

  ```
  WIRELESS_SHOW(crypt, discard.code, fmt_dec);
  WIRELESS_SHOW(beacon, miss.beacon, fmt_dec);
  ```

  Similarly in Openssh, there is a macro called "SPLAY_PROTOTYPE(name, type, field, cmp)" defined in "openssh-3.9p1/openbsd-compat/sys-tree.h", which is used to generate prototypes and inline functions. Functions like "mmtree_RB_INSERT" and "mmtree_RB_REMOVE" which are defined in "openssh-3.9p1/monitor_mm.o" are generated by the above macro.
- **Override a Function Definition**:
  Our analysis revealed an interesting usage of macros in the studied systems. For example, in Postgres the entity "sslow" is defined in file "src/backend/regex/engine.c" as follows:

  ```
  #define slow sslow
  ```

  In this example, the rest of the functions in the file refers to a function "slow" which is defined in the file, but no function "sslow" is defined. Instead the "`#define`" is used to override the name of the function "slow" to "sslow". It appears that the macro is used to simulate inheritance in C. To implement a different function, the only change needed is to alter the defined value ("sslow") and the code will call the new function.

Comparison of Dependencies
After the *Entity Comparison* phase, we can now compare the dependencies between the entities generated by both extractors. The Dependency Comparison phase is divided into 3 steps:

- **Step 0 – Initial State:** We compare the dependencies produced by both extractors without performing any modifications to the data.

Description	Openssh	Postgres	Linux
Total initial *LDX* entities	1,529	6,965	65,064
Entities that only *LDX* has	88	296	16,039
Differences due to *LDX*' incorrect output			
→ Static variable inside functions	78	90	1,008
Differences due to Build Process			
→ Files generated during the build	0	2(+3)	1,745(+1,747)
→ Including non-standard files	0	61	0
Differences due to Preprocessor			
→ Macro defined entities	8	92	12251
→ Overriding a function definition	2	47	13(+727)
Differences due to *ctags*' limitations			
→ Complex conditional directives	0	3	0
→ Confusion in determining name of an entity	0	0	306

Table 3: Entity Comparison Statistics for Openssh, Postgres and Linux (Numbers in Brackets Indicate Differences that were Categorized Through Manual Analysis of the Source Code and the Binary Object File)

- **Step 1 – Common Entities:** We focus our analysis on dependencies between the common entities between both extractors. We remove all dependencies that do not originate or terminate in an entity which is in common between both extractors using the results of our entity comparison phase. The intuition behind this step is that difference is likely due to an entity that existed in a file that was not part of the studied executable since *CTAGX* analyzes all files in the source code directory without taking into account the build configurations.
- **Step 2 – Macro Expansion:** Since *CTAGX* analyzes the code before the expansion of macros and *LDX* analyzes code after their expansion, we use the *CTAGX* facts to simulate the expansion of macros. We expand the macros in the *CTAGX* facts. We then compare the dependencies between the *CTAGX* facts after macro expansion and the *LDX* facts.
- **Step 3 – Macro Expansion Cleanup:** The expansion of macros adds a large number of dependencies. For example, if there are several conditional `#ifdef` branches inside a function, we would expand all the macros inside these branches during the prior step. In this step, we go through the *CTAGX* dependencies and we remove all dependencies that were added during the macro expansion and that did not exist in the *LDX* facts in Step 2.

Steps	*CTAGX* only	Common	*LDX* only
Initial State	47.78%	38.72%	13.50%
Step 1	16.43%	78.97%	4.60%
Step 2	17.35%	81.87%	0.78%
Step 3	16.45%	82.76%	0.79%

Table 4: Dependency Comparison Statistics for Openssh

Tables 4, 5 and 6 display the percentage for Initial State and the 3 steps in our analysis along three criteria:

Steps	*CTAGX* only	Common	*LDX* only
Initial State	70.93%	20.89%	8.18%
Step 1	10.78%	74.16%	15.06%
Step 2	17.10%	82.78%	0.13%
Step 3	11.23%	88.63%	0.14%

Table 5: Dependency Comparison Statistics for Postgres

Steps	*CTAGX* only	Common	*LDX* only
Initial State	91.26%	4.74%	4.00%
Step 1	50.15%	28.81%	21.05%
Step 2	86.56%	10.43%	3.01%
Step 3	56.03%	34.12%	4.85%

Table 6: Dependency Comparison Statistics for Linux

1. **CTAGX only:** Percentage of dependencies in *CTAGX* but not in *LDX* in comparison to the total number of dependencies generated by both extractors.

2. **Common:** Percentage of dependencies that are common between both *LDX* and *CTAGX* in comparison to the total number of unique dependencies generated by both extractors.

3. **LDX only:** Percentage of dependencies in *LDX* but not in *CTAGX* in comparison to the total number of dependencies generated by both extractors.

Figures 6(a), 6(b), 6(c) are plots of the data in Tables 4, 5, 6 for Openssh, Postgres, and Linux respectively. The results of our dependency comparison phase reveal that:

- *LDX* misses a large number of *CTAGX* dependencies. These missing dependencies are due to the fact that *CTAGX* analyzes and extracts dependencies from all conditional paths (`#ifdef`) inside a function, whereas

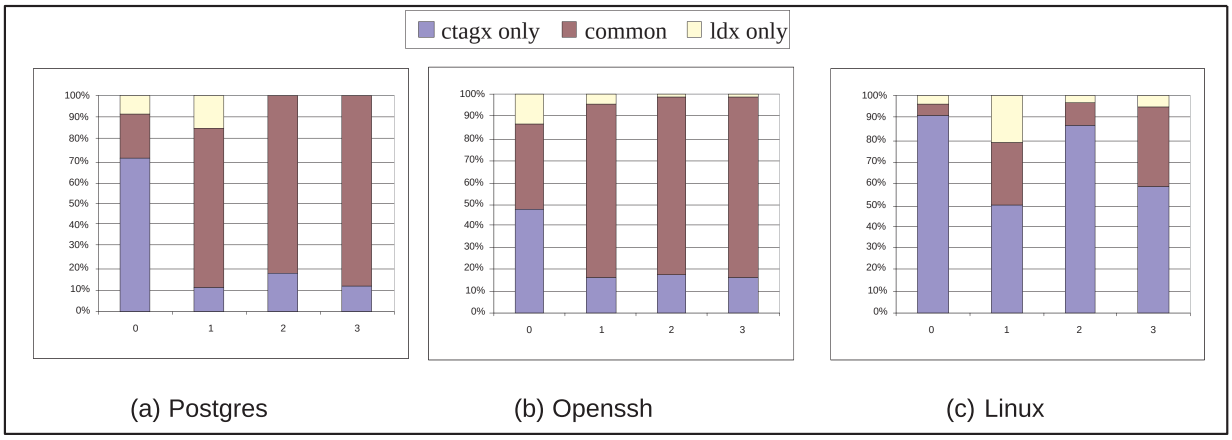

Figure 6: Dependency Statistics for Postgres, Openssh and Linux For each Comparison Step

LDX can only recover information for a single conditional path which corresponds to a particular build configuration.

- For Openssh and Postgres, there is less than 1% of the *LDX* dependencies that *CTAGX* cannot recover. However, there are about 5% of the *LDX* dependencies in Linux that *CTAGX* cannot retrieve. Through manual analysis of the source code and the generated binary code, we discovered that in several cases *LDX* outputs data that corresponds to the code after compiler optimization. These cases account for the 5% missing dependencies in Linux and the 1% in the other two systems. For example, if a function (FuncA) references static variable (struct_test) which contains a reference to a function (FuncB), the compiler might optimize out the access to the static variable and the *LDX* facts would only show a dependency from function (FuncA) to function (FuncB). The code snippet below illustrates this example.

```
int FuncB() {
        printf("hello world\n");
        return 0;
}

static struct func_ptr_struct {
        unsigned int test_int;
        int (*test_ptr) ();
} struct_test;

int FuncA() {
        struct_test.test_int = 7;
        struct_test.test_ptr = FuncB;

        if ((struct_test.test_ptr)() == 0) {
                return 0;
        }
        return 1;
}
```

There are many similar examples in the studied systems in particular in Linux. For example, *LDX* indicates that function "intel_i810_alloc_by_type" which is defined in "linux-2.6.1/drivers/char/agp/intel-agp.c" depends on "agp_generic_mask_memory". Examining the source code we note that the function de-references a member field in a static struct which contains a pointer to "agp_generic_mask_memory".

- *CTAGX* only dependencies are larger in Linux in comparison to the other studied systems. This is due to the fact that Linux has a large number of build configuration which are defined using macros and `#ifdef`'s which *CTAGX* analyzes but *LDX* is not capable of analyzing.

5 LESSONS LEARNED

We learned several lessons from our comparative experiment. We now discuss these lessons in detail.

Lesson 1 – Facts produced by an object code extractor don't reflect the source code level dependencies seen by developers

Although an object code extractor may be easier to implement, software developers should be careful when examining the dependencies produced by it. An object code extractor produces facts based on entities and relations that a developer is likely never to encounter when browsing or editing the code in a source code browser. This occurs because such an extractor produces facts based on code that has been preprocessed and optimized.

Lesson 2 – Facts produced by a source code extractor reflect the source code level dependencies seen by developers and contain most of the information produced by an object code extractor

A source code extractor performs its analysis on the source code as seen by a software developer. Developers are likely to be more comfortable examining facts produced by a source code extractor since they are based on the source code as shown in the development environments used by developers. Our experiment has shown that by expanding macros, facts produced by a source code extractor contain almost all of the facts produced by an object code extractor.

Lesson 3 – Both types of extractors are valuable for developers
For software developers trying to understand a software system both types of extractors would be valuable. Although an object code extractor would produce dependencies that are not visible to developers at the source code level, the dependencies produced by an object code extractor would reveal hidden dependencies that may not be clear due to macros and complex compilation directives. It would be interesting to explore annotating the more complete facts produced by a source code extractor to indicate which macros are expanded, which conditional define branches are followed, and which hidden dependencies exist for a particular build configuration.

Lesson 4 – The build time view should be explored when analyzing large software systems
Our results indicate that several files and entities are defined during the building of a software system. A good knowledge of the build process of a software system is valuable. The build time view of a software system, presented in [8], should be explored during architecture analysis since it is likely to reveal in many cases unexpected dependencies.

Lesson 5 – Heuristic parsing is a useful technique for analyzing large software systems
CTAGX and *ctags* use a number of heuristics to perform their analysis on incompilable or incomplete source code. Although in our presented experiment we did not discuss the results of using *CTAGX* on code with syntax errors, our experiment showed that *CTAGX* can analyze code without needing a makefile and that the produced facts represent almost all the facts produced by the *LDX* extractor which performs its analysis using well defined grammars and complete build instructions.

Lesson 6 – Using our comparison process to evaluate other extractors
Although the main purpose of our comparison process, outlined in Figure 4, was to compare the facts produced by the two particular extractors, we found that our process is valuable in improving the quality of the studied extractors. Others can use our comparison process for testing and improving their extractors. Throughout our analysis, we uncovered many bugs in *CTAGX* and *LDX* that have been corrected and verified as a result of using our comparison process.

Lesson 7 – The gap between the pre and post processed source code is significant for large software systems
During our dependency comparison phase, we examined closely the number of macro expansions needed in the studied software systems. We found that a single macro was expanded as many as 6 levels in a software system. Table 7 shows the percentage of macros that are expanded during each level of macro expansion for Openssh, Postgres, and Linux. For example in Openssh, all macros are expanded up to 2 levels deep: about 20% of all macros are expanded once, and about 8% of all macros that exist in the source code are expanded twice. The same information is shown graphically in Figure 7. It takes 2, 3 and 6 levels to expanded all macros in Openssh, Postgres and Linux respectively. This large number of levels increases the gap between the developer's view of the source code and the code that is actually executed. It is interesting to note that the macro expansion rate for the first level is not 100%, since in some cases macros are defined in header files to avoid circular includes. It would be interesting to explore the percentage of used macros for each build configuration and to measure the amount of dead macros that are not used in any build configuration. These dead macros could be refactored and removed from the code.

Level	Openssh	Postgres	Linux
1	20.21%	49.96%	46.80%
2	7.77%	32.05%	33.60%
3		21.37%	27.07%
4			20.99%
5			18.43%
6			17.42%

Table 7: Levels of Macro Expansion for Each Studied Software System

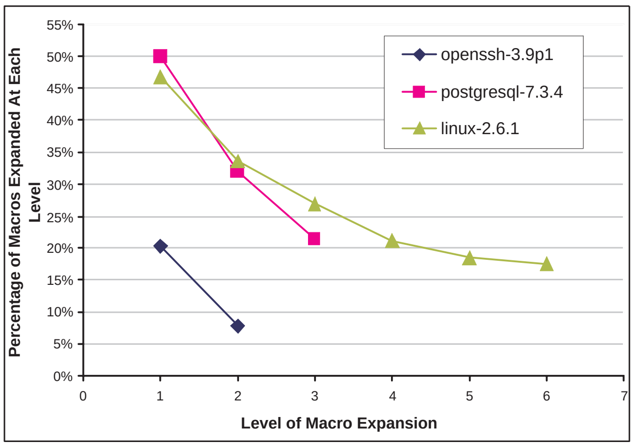

Figure 7: Macro Expansion Statistics

6 CONCLUSION

To overcome the lack of documentation, software developers can recover the software architecture from a system's implementation. We pointed out that there are many types of extractors that could be used to extract dependencies. These extracted dependencies are lifted to recover the architecture from the implementation. We focused on two types of extractors: a source code and an object code extractor. In contrast to previous work which compared extractors [1, 2, 7], we focused on performing our analysis on much larger open source case studies while ensuring that we thoroughly investigated the differences between the facts produced by each type of extractor to have a better understanding of the reasons behind the differences. In particular, we explored whether these differences are due to implementation bugs, or if they are due to limitations of the used extraction technique.

Our experimental results show that both types of extraction techniques are useful in recovering the dependencies between entities in a large software systems. We as well showed that the use of macros causes a significant gap between what software developers see in their code editors and the actual code that executes. This gap translates to a large difference between the facts generated by source and object code extractors. Furthermore, the gap causes the appearance of unexpected dependencies that are likely to introduce bugs in the software system. Finally, our comparison process has been useful in improving the quality of the studied extractors and could be adopted by other developers of extractors.

REFERENCES

[1] M. N. Armstrong and C. Trudeau. Evaluating architectural extractors. In *Proceedings of the 5th Working Conference on Reverse Engineering*, pages 30–39, Honolulu, HI, Oct. 1998.

[2] B. Bellay and H. Gall. A Comparison of four Reverse Engineering Tools. In *Proceedings of the 4th Working Conference on Reverse Engineering*, Amsterdam, The Netherlands, Oct. 1997.

[3] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a Case Study: Its Extracted Software Architecture. In *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, USA, May 1999.

[4] Exuberant Ctags. Available online at `http://ctags.sourceforge.net`.

[5] T. R. Dean, A. J. Malton, , and R. C. Holt. Union schemas as a basis for a C++ extractor. In *Proceedings of the 8th Working Conference on Reverse Engineering*, Stuttgart, Germany, Oct. 2001.

[6] A. E. Hassan and R. C. Holt. Architecture Recovery of Web Applications. In *Proceedings of the 24th International Conference on Software Engineering*, Orlando, Florida, USA, May 2002.

[7] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology*, 7(2):158–191, 1998.

[8] Q. Tu and M. W. Godfrey. The Build-Time Software Architecture View . In *Proceedings of the 17th International Conference on Software Maintenance*, pages 398–408, Florence, Italy, 2001.

[9] J. Wu and R. C. Holt. Linker-Based Program Extraction and Its Uses in Studying Software Evolution. In *Proceedings of the International Workshop on Unanticipated Software Evolution*, pages 1–15, Barcelona, Spain, Mar. 2004.