An Empirical Study of `goto` in C Code

Meiyappan Nagappan*, Romain Robbes†, Yasutaka Kamei‡, Éric Tanter†,
Shane McIntosh§, Audris Mockus¶ and Ahmed E. Hassan§
*Department of Software Engineering, Rochester Institute of Technology, USA
†PLEIAD Lab, Computer Science Dept (DCC), University of Chile, Chile
‡Principles of Software Languages Group (POSL), Kyushu University, Japan
§Software Analysis and Intelligence Lab (SAIL), Queen's University, Canada
¶Department of Electrical Engineering and Computer Science, University of Tennessee-Knoxville, USA
*mei@se.rit.edu, †rrobbes@dcc.uchile.cl, ‡kamei@ait.kyushu-u.ac.jp, †etanter@dcc.uchile.cl,
§shanemcintosh@acm.org, ¶audris@mockus.org, §ahmed@cs.queensu.ca

PeerJ PrePrints

***Abstract*—It is nearly 50 years since Dijkstra argued that `goto` obscures the flow of control in program execution and urged programmers to abandon the `goto` statement. While past research has shown that `goto` is still in use, little is known about whether `goto` is used in the unrestricted manner that Dijkstra feared, and if it is 'harmful' enough to be a part of a post-release bug. We, therefore, conduct a two part empirical study - (1) qualitatively analyze a statistically representative sample of 384 files from a population of almost 2 million C programming language files collected from over 11K Github repositories and find that developers use `goto` in C files for error handling ($80.21 \pm 5\%$) and cleaning up resources at the end of a procedure ($40.36 \pm 5\%$); and (2) quantitatively analyze the commit history from the release branches of six OSS projects and find that no `goto` statement was removed/modified in the post-release phase of four of the six projects. We conclude that developers limit themselves to using `goto` appropriately in most cases, and not in an unrestricted manner like Dijkstra feared, thus suggesting that `goto` does not appear to be harmful in practice.**

I. Introduction

In the March 1968 issue of the *Communications of the ACM*, Edsger W. Dijkstra published his observations of the problems caused by using `goto` statements in programs, titled *Letters to the Editor: Go To Statement Considered Harmful* [1]. This is one of the many works of Dijkstra that is frequently discussed by software practitioners [2] and researchers alike (more than 1,300 citations according to Google Scholar and almost 4000 citations according to ACM Digital Library as of Aug 15, 2014). This article has also resulted in a slew of other articles of the type *X considered harmful*, where *X* has been instantiated to global variables [3], polymorphism [4], fragmentation [5], among many others. In fact, Meyer claims that as of 2002, there are thousands of such articles, though most are not peer-reviewed [6].

Indeed, Dijkstra's article [1] has had a tremendous impact. Anecdotally, several introductory programming courses instruct students to avoid `goto` statements solely based on Dijkstra's advice. Marshall and Webber [7] warn that when programming constructs like `goto` are forbidden for long enough, they become difficult to recall when required.

Dijkstra's article on the use of `goto` is based on his desire to make programs verifiable. The article is not just an opinion piece; as Koenig points out [8], Dijkstra provides strong *logical* evidence for why `goto` statements can introduce problems in software.

Since Dijkstra's article, there have been several authors who have theoretically analysed the harmfulness (and sometimes benefits) of `goto` (Section II), but there are seldom any empirical studies on them. In the recent past, two studies examined the use of `goto` in C code for error-handling. However, they focus on improving the error handling mechanism [9], [10], and not to characterize the use of `goto` statements in practice by developers or examine their harmfulness.

In this paper we empirically examine the use of `goto` statements in modern software projects by software developers after ascertaining that `goto` is indeed widely used. More precisely, we carry out a two-dimensional case study of C source code files from software repositories- a broad qualitative study on a statistically representative sample of code from 11,627 repositories hosted by Github [11] followed by a focussed quantitative study on specific releases of six OSS projects to address the following research questions:

- **RQ1:** *What do developers use `goto` statements for?*
  We find that `goto` statements are used for 5 main purposes. Among these, the most prevalent are *handling errors* and *cleaning up resources at the end of a procedure* ($80.21 \pm 5\%$ and $40.36 \pm 5\%$ respectively).
- **RQ2:** *Are `goto` statements involved in post-release bugs?*
  We find that no `goto` statement is removed/modified as part of post-release bug fixes in four of the six projects. In the remaining two projects, `goto` was removed/modified only in a dozen bug fix commits.

II. Background and Related Work

Dijkstra thought that the title of the article [1] misrepresented the actual argument he tried to make. He explains that his article is referenced *'often by authors who had seen no more of it than its title'*.[1] Therefore, we start with a detailed discussion of the arguments given by Dijkstra in order to provide context to the reader, and then discuss related work that examined Dijkstra's communication.

[1] All italicized text within single quotes in this section are direct quotes from Dijkstra.

PeerJ PrePrints | http://dx.doi.org/10.7287/peerj.preprints.826v1 | CC-BY 4.0 Open Access | rec: 11 Feb 2015, publ: 11 Feb 2015

A. Background

Dijkstra begins the article by stating that in his observation of programmers and programs, the use of `goto` statements has *'disastrous effects'*. This is the reason why he believes that `goto` statements should not be used in high level languages, in which most current software is written. He then goes on to explain why the `goto` statements can have disastrous effects and what those effects might be.

He prefaces his reasoning with two remarks:

1) The main responsibility of the programmer is to know and understand the execution of their program (*'process taking place under the control of a program, which is delegated to the machine'*).
2) Due to the intellectual limitations inherently present in us, we are unable to *'visualize processes evolving in time'*. Hence, programmers should try their *'utmost to shorten the conceptual gap between the static program and the dynamic process'*.

The objective is for a programmer to be able to establish clear assertions about the state of the executing program at each line of code. Dijkstra then observes that the use of `goto` statements in a program can make it difficult to determine the exact set of statements that has been executed before reaching a given label block. For instance, consider the following example:

```
1  void fun(int x){
2   back:
3     if (condition1) {
4       x = x + 1;  goto next;
5     }
6     else if (condition2) {
7       x = x + 2; goto back;
8     }
9     x = x + 3
10  next:
11    print(x);
12    if (!done) goto back;
13 }
```

The function body declares two labels, `back` and `next`. There are three ways one could execute the `back` label block: when first entering the function, because of `condition2`, or because of `!done` in the `next` label. Similarly, the `next` label block can be entered either because of `condition1`, or because both conditions were false. The statement on line 9 may or may not have been executed before entering `next`. Making assumptions about the state of `x` at different program points is quite complex.

Dijkstra explains that, in a language with procedures, one can characterize the progress of a process with a *coordinate system* that consists of a sequence of textual indices—the sequence of procedure calls (the call stack) followed by a single point in the currently-executing procedure. With the inclusion of repetition clauses such as `while`, one needs an additional dynamic index. Reasoning is supported because the repetition structure, like recursion, is amenable to inductive reasoning. In any case, the crux of the argument is that the coordinate system is independent of the programmer. In contrast, in the case of `goto` statements*'it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress'*. In essence, *'The go to statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program.'*

B. Related Work

In this subsection, we present related work (essays and articles) done by others who have either carried out their own analysis on the use of `goto` statements, or have critically analyzed Dijkstra's article.

Analyzing the use of `goto` statements: The article by Knuth [12] is probably one of the earliest works analyzing `goto` statements, since Dijkstra's article was published. Knuth clearly prefaces the article by stating that only his *'opinions'* are expressed. His goal for the article is to provide a fair discussion of both the pros and cons of `goto` statements. He believes that the discussion should not be about eliminating `goto` statements but about program structure. He points out that in fact several cases of `goto` elimination is merely replacing `goto` statements with other *'euphemisms'*. Knuth [12] believes that `goto` statements must not be used for error exists or loop creation, but only used to eliminate recursion, boolean variables, and co-routines. He even quotes Dijkstra to drive home the point: *'Please don't fall into the trap of believing that I am terribly dogmatical about [the go to statement]. I have the uncomfortable feeling that others are making a religion out of it, as if the conceptual problems of programming could be solved by a single trick, by a simple form of coding discipline!'*. More recently Saha *et al.* examine how `goto` statements are used for error-handling and propose approaches to improve the error-handling mechanism in C code [9], [10]. When empirically evaluating their approach, they find that `goto` is still extensively used in system projects written in the C programming language. However, the studies by Saha *et al.* [9], [10] are not about examining all the different ways in which developers use `goto`, but just focussed on the use of `goto` for error-handling.

Analyzing Dijkstra's article: In his retrospective essay, Tribble [13] provides an annotated version of Dijkstra's article, and also analyzes the relevance of the article in the year 2005. He questions whether `goto` statements are even needed, when developers are using languages that were developed well after Dijkstra's article was published. He concludes that (in contrast to Knuth [12]) certain complex `goto` statements cannot be avoided if the programming language does not provide dedicated constructs for exiting nested loops or for error handling. Since programming languages like C do not have an explicit error handling mechanism (exception handling) like C++/Java, the developers who write their software in C have to resort to using `goto` statements. In the qualitative empirical analysis in RQ1 (Section V), we examine if developers are indeed using the `goto` statements for error handling and for exiting out of loops among other purposes.

There are several references in the literature [14], [8], [12] about heated discussions and analysis on Dijkstra's article

PeerJ PrePrints

PeerJ PrePrints | http://dx.doi.org/10.7287/peerj.preprints.826v1 | CC-BY 4.0 Open Access | rec: 11 Feb 2015, publ: 11 Feb 2015

among practitioners. One such recorded discussion [2] can be found on StackOverflow, a community 'question and answer' website for developers. The discussion brings about similar points as presented above, and there is consensus that `goto` statements need not be eliminated entirely, but just the *'unconstrained'* use of it should be avoided. Another such discussion was among well-renowned Linux kernel developers—Robert Love, Rik van Riel, and Linus Torvalds [15]. Their position is that `goto` can be used when there is a real need for it and that, if used carefully, `goto` is not harmful.

Unlike prior work, which have taken logical, argumentative, and example-based approaches, we use an empirical one to identify, classify and quantify the different actual uses of `goto` statements in real-world C code, and examine if `goto` is harmful enough that developers removed/modified them as part of post-release bug fixes in a software project. While the previous work is in no way incorrect or insufficient, we believe that adding empirical evidence can only further help the discussion.

III. Data Source

In order to examine empirically the use of `goto` statements along the two research questions we have, we used two different data sources. In both datasets, we restrict our analysis to just C files, since (1) it is a widely-used language; and (2) it provides the `goto` construct; Additionally, we wanted to examine the use of `goto` statements without any confounding factors based on different programming languages. By restricting to just one programming language we can place the results in context, and have more confidence about the conclusions with respect to software projects written in C. Below we describe the data source used for each research question and the reasons for choosing them.

Data Source for RQ1: In this research question, we want to understand the purpose of `goto` statements as used by a variety of developers in their code. Hence studying the programming practices of developers from a small set of projects may not be sufficient. Therefore, we use the source code in the hundreds of thousands of software projects mined by one of the coauthors [16]. We use a snapshot of the project's software repositories (that are being continuously mined) from January 2013. We restrict our analysis to software projects that have their source code repositories on the Github hosting service. This is because, Github hosts several million repositories from several hundreds of thousands of developers. Hence we will be able to examine developer practices with respect to `goto` statements among a wide variety of developers. From this collection of software projects we extract all projects that have C files.

Not all repositories on Github represent actual software projects. We filtered out projects with less than 10 files written in any programming language to eliminate a bulk of non-software projects, and very small software projects. We chose the cutoff at 10 as it appeared to provide a balance between eliminating most irrelevant repositories while leaving us with a sizable sample of repositories. While this sample contains some repositories unrelated to software development identifying them automatically on the scale of Github is a research topic by itself, well beyond the scope of this paper.

TABLE I: Overview of the studied systems (RQ2).

Product	Domain	Version	Tag name	Lines of code
ClamAV	Antivirus	0.98	clamav-0.98	2,043,360
GIMP	Image Manipulation	2.8.0	GIMP_2_8_0	960,106
GhostScript	Interpreter	9.01	ghostscript-9.01	1,274,249
OpenLDAP	Directory Access Protocol	2.4.6	OPENLDAP_REL_ENG_2_4_6	303,917
Postgresql	Database	9.3.0	REL9_3_0	963,900
VTK	Visualization Toolkit	5.10.0	v5.10.0	2,910,353

The filtered sample has 11,627 projects with C code. A typical empirical software engineering study examines under 10 projects [17], and this larger sample should help with the generalizablity of our conclusions (albeit in the context of source code written in C and hosted on Github).

In the final data cleaning step we filtered out files we identified as automatically generated. Such files are not directly maintained, and, consequently, do not shed light on how developers are using `goto` statements. To develop the filtering procedure, we manually examined a random sample of files with hundreds of `goto` statements to identify if any of them were generated. Every generated file had the word 'generate' in the comments at the beginning of the file. Our filtering, therefore, searches for the word 'generate' in each file. If the term occurs, we then filter the file from the dataset. The threats that arise due to not removing generated files are discussed in Section VII. The cleaned sample contains 2,150,387 files written in the C programming language from 11,627 projects hosted on Github.

Data Source for RQ2: We would like to know what impact `goto` statements have on the incidence of post-release bugs: a popular proxy for software quality [18] because not all bugs are of equal concern. Bugs that could be fixed in development and testing are of much lesser concern than bugs that escape quality assurance steps and affect software users. We, therefore, focus on post-release bugs, which are not detected until after the software is made available to end users.

To identify post-release bugs, we consider code changes that occur on or that have been merged into the release branch of a studied system. Such changes are often the result of bugs found after the release of software and these changes are pushed to customers as part of a software update. As a result, changes that land on a release branch after a release are more strictly controlled than a typical development branch to ensure that only the appropriately triaged high-priority fixes land in maintenance releases. In other words, the changes that we study correspond to the maintenance of official software releases.

Unfortunately, the overwhelming majority of projects examined in RQ1 do not contain carefully triaged release-related changes. Hence, to address RQ2, we selected six software

PeerJ PrePrints | http://dx.doi.org/10.7287/peerj.preprints.826v1 | CC-BY 4.0 Open Access | rec: 11 Feb 2015, publ: 11 Feb 2015

projects primarily implemented in C with well-established branching policies for deeper analysis. These projects are selected from a variety of domains, ages, and sizes to help with the generality of our conclusions. Some of these projects like GIMP are used by millions of end users [19], while others like Postgresql are used by prominent industrial users such as Nasa [20], and Instagram [21]. Table I provides an overview of the studied systems.

IV. Preliminary Analysis: Do developers use `goto` statements in their source code?

A. Motivation

Before we examine *why* developers use `goto` statements, we first need to determine *if* they even use `goto` in the source code of their software. Even though, past studies have shown that `goto` statements are still in use [9], [10], we want to examine if our dataset of diverse projects have `goto` statements in them and ascertain the true extent of `goto` usage.

B. Approach

Among the 2,150,387 source code files used in our study, we determine which ones have a `goto` statement in it. We do this by using the grep functionality for searching and use the regular expression

```
([ \t]+goto|^goto)[ \t]+.*;
```

We determine not just if a file has a `goto` statement, but also how many `goto` statements exist in each file.

C. Results

Considerable use of `goto` at the file level: We find that 246,657 out of the 2,150,387 files (or 11.47%) examined in our study have at least one `goto` statement. Both the raw number and the percentage of files that have a `goto` statement indicates that the use of `goto` statements by developers who work on software projects written in C and hosted in Github, are quite common. The result agrees with the results of Saha *et al.* [9] – who found that there are almost 20K functions in Linux-2.6.34 that have a `goto` statement. We also find that most files have very few `goto` statements. In fact 14.43% of files with `goto` had only one `goto`. However, there are those rare occasions where there exists a file with more than 100 `goto` statements (0.55% of the files with `goto`).

More than one-fourth of the projects used a `goto`: In order to check if all the files with `goto` are just found in a few projects, we calculate the distribution of `goto` statements at the project level. We find that 3,093 out of the 11,627 projects (or 26.60%) have at least one file with a `goto` statement. We also find that more than half the projects have about 20% of the files that have at least one `goto` statement.

> *A considerable number of projects have files with `goto` in them. Thus despite the popularity of Dijkstra's case against `goto`, developers use `goto` statements considerably.*

D. Discussion

When examining the path hierarchy and names of files with and without `goto` statements in them, we noticed that several of them looked like system code, i.e. they had the words linux/android/kernel/driver (which is consistent with the findings of Saha *et al.* [9], who report that more than half of the `goto` statements are in the driver directory of Linux-2.6.34). To better characterize the context of use of `goto` statements, we examine the number of "system files" (source code files for hardware drivers, operating system kernel etc.) in the dataset. A search for the keywords linux/android/kernel/driver in file names revealed that 454,670 files (21.14%) were system files. Of course, we could have missed some system files, but due to the size of the dataset, we were restricted to using such a heuristic, whose precision was verified with a manual analysis in RQ1 (Section V). Among the files that had a `goto` statement, we found that 169,523 (or 68.72%) were system files. Thus, we find that there is a greater concentration of `goto` statements in system files than non-system files.

V. RQ1: What do developers use `goto` statements for?

A. Motivation

Beyond the raw frequency of `goto` usage, we were also interested in *how* developers are using them. This is because not all usages of `goto` are created equal. Some `goto` usage may be part of relatively harmless patterns, while others may be much more nefarious [12], severely hampering the readability of the source code. A related concern is that `goto` statements may be used in various domains, and may be used differently in these domains.

B. Approach

We, therefore, conducted a qualitative study of the usages of gotos on two levels: the file level, to understand the domain in which `goto` statements were used, and figuring out whether many false positives were due to generated code; and the function level, to understand patterns of usage of `goto` statements, their interactions, and their impact on the function comprehensibility. We performed the latter analysis at the function level, since the scope of the `goto` statements and the labels that they refer to are at the function level. We discuss the process for our qualitative study in detail below (as it will help us and the reader place the results and conclusions in context).

1. Sample Selection: The population of C files that we examined for the presence of `goto` statements in RQ1 is 2,150,387. However, we want to study the patterns of usage of `goto`. Hence our true population is the set of files with `goto` statements, i.e., 246,669 C files. We split this population into two buckets - files with more than 5 `goto` statements, and files with 5 or less `goto` statements, where 5 is the median number of `goto` statements in a file. From each bucket we pick 192 files at random, to end up with a total of 384 C files with `goto` statements. By examining 384 files from 246,669 files, we achieve a confidence level of 95% with a confidence

PeerJ PrePrints | http://dx.doi.org/10.7287/peerj.preprints.826v1 | CC-BY 4.0 Open Access | rec: 11 Feb 2015, publ: 11 Feb 2015

TABLE II: A taxonomy of the different purposes and properties of goto statements.

	Name	Description	Rationale
File Level	Generated	The source file containing the goto statement was generated automatically.	We are only interested in goto statements that programmers introduce manually.
	Long jumps	Non-local jumps performed with longjmp to a context captured with setjmp.	This has potentially much more serious consequences than regular (local) goto statements.
	Domain	The domain to which the file belongs to.	Knowing the domain of each file, lets us understand the type of systems that are frequently using goto statements.
Purpose - Function Level	Error	To handle exceptions and errors. Tagged when the code in the label is for error handling.	Since C does not have exception handling constructs like try/catch in Java. Also discussed in the literature [13], [12].
	Cleanup	To handle memory de-allocation and other cleanup activities. Tagged when the code in the label is executed whether or not the goto statement is executed.	Since C does not have a cleanup construct like finally in Java.
	Control-Exit	To exit out of a nested loop.	Since C does not have break to labels. Also discussed in the literature [13], [12].
	Loop-Create	Unlike Control-Exit, this is when a goto statement is used to create a loop.	A priori unnecessary because C has repetition clauses (while, do, for, ...).
	Spaghetti	When a goto statement exists in the code inside a label of another goto statement.	This is the kind of goto statements that all prior work has argued against [1], [13], [12].
Properties - Function Level	Single	Only one goto statement per label.	According to Dijkstra, while less harmful, a single goto per label can still cause issues [1].
	Multiple	Many goto statement per label.	A source of severe issues according to Dijkstra [1].
	Forward	Jump to a label that is located after the goto statement.	Easier to track by programmers because it follows the natural order of reading code.
	Backward	Jump to a label that is located before the goto statement.	Harder to follow because it requires going back in the procedure definition.
	#goto statements per LOC	The number of goto statements per line of code in each function that is examined.	This will allow us to see if there is a relationship between the number of goto statements and the size of a function.
	Stacked labels	Multiple labels that follow each other such that execution flows from one label block to the next.	A mixed blessing: jumped-to code is localized, but there are multiple entry points in the aggregate block, making it hard to know which statements are actually executed.
	#statements in label block	Number of statements in the label block.	An (imperfect) estimate of the complexity of the functionality implemented in the label.

interval of 5% [22]. We use this sample of 384 files to perform the qualitative analysis at the file level.

In order to carry out our qualitative analysis at the function level, we take the sample of 384 files chosen for the qualitative analysis at the file level, and follow a procedure similar to that of Gallup polls [23]. For example, when polling adults in the United States(US) to determine election results, Gallup randomly calls households across the US, and picks one individual in the household randomly to interview. Similarly, in order to carry out the study at the function level, we randomly pick one function (an individual person in the gallup scenario) from each file (an individual household in the Gallup scenario). Thus we now have 384 randomly chosen functions with goto statements and with the same confidence level and confidence interval. We then manually tag the sample data (384 randomly chosen files and functions).

2. Identifying Tags from Subset of Sample: From this sample of 384 files, we came up with a preliminary classification of manual tags at the file and function level, based on separate inspection of two pairs of 10 files, by authors 1 and 2, and 4 and 5. The identified tags at the file level and function level are described in Table II. At the function level, we further split the tagging into two categories - properties of goto statements and the purpose of goto statements.

3. Iterative Tagging of Complete Sample: Then each of the four authors manually tagged a mutually exclusive subset of files, while constantly discussing with the others for clarifications and refinements of the tags. Each author went through the tags assigned to each file and function at least twice. Once we tag each file and function in the sample, we examine the frequency of the tags in order to understand the usage of goto. We present these results in the next subsection.

C. Results at the file level

After manually tagging the 384 files at the file level, we arrive at the following findings:

Only 1.5 % of the files were filtered for noise: Out of the 384 files, only five files were automatically generated files. These files were generated by a parser generator like YACC (Yet Another Compiler Compiler). We also found one case of a file that was submitted to the ACM International Collegiate Programming Competition. Even though the file submitted to the competition is a software program, we remove it from further analysis, since we know that it was not developed for a software project. Hence, we did not manually continue tagging with a total of six files. Therefore, the fraction of irrelevant files present in our sample is $(1.5 \pm 5)\%$.

A large portion of files (85%) were system or network files: We found that 312 out of the 384 files were in the domain of 'systems programming'. We define that a file is in the domain of 'systems programming' when it is associated with the Linux Kernel or any other OS, like the BSD or Android or Sun OS. We also associate all driver related files in the domain of 'systems programming'. There were also 13 files

PeerJ PrePrints | http://dx.doi.org/10.7287/peerj.preprints.826v1 | CC-BY 4.0 Open Access | rec: 11 Feb 2015, publ: 11 Feb 2015

PeerJ PrePrints

that were associated with ‘Networking’. Thus overall we have 325 out of 384 files in the ‘Systems and Networking’ domain. This translates to almost $85 \pm 5\%$ of the files in the dataset. There could be two possible reasons for this - (a) C files in github are mostly system/networking purposes, or (b) `goto` is used more in systems/networking files.

Some of the other domain types that we identified were: Image Processing/Multimedia/Videogames/Web/UI Framework related files with a total of $7.28 \pm 5\%$, and PL/DB/HPC/Scientific/Security related files with a total of $5.72 \pm 5\%$.

Only one case of `setjmp`: Apart from `goto`, which is restricted to jumps local to a function, C provides `setjmp` to perform arbitrary jumps between execution contexts. This is arguably the worst case of a `goto`-like construct because it compounds the effects of `goto` statements. We found that there was only one file in which the command `setjmp` was used, so we can safely assume that developers seldom use this in their programs.

Summary: Our manual analysis at the file level led to several findings: it showed us that gotos are used predominantly in systems and networking code, but also in other type of files as well. It also gave information alleviating concerns on some possible threats to validity: extremely few of the files were noise, giving confidence in our filtering mechanism; a single file used the a non-local jump (`setjmp`).

D. Results at the function level—Basic properties

We start our investigation of the functions we tagged by reporting on the distribution of basic properties on the number of `goto` statements, labels, and the length of label blocks.

Most functions have few `goto` statements and labels: We find that at least 25% of the functions in the sample have 1 `goto` statement. The median is 3, and the top 25% of functions have at least 5 gotos. We also find some outliers, functions with 10 or more gotos, and one extreme outlier that has more than 50 gotos. Unsurprisingly, there are less labels than `goto` statements. The median function has just one label, i.e, at least half of the functions in our sample have only one label in their body. The upper quartile is 2, and there a few functions with 5 or more labels. Overall, both distributions show that the majority of functions have few `goto` statements and labels.

Usually, few lines are in the label blocks: We summed the amount of lines of code under each label of the functions in our corpus. We found that the median lines of code in the block of code under a label was 4, and the 75th and 90th percentile was 8 and 36 LOC respectively. We found that in most of the non-trivial cases, the code in the label block had a line of code for clean up, a line of code for printing an error message, and a line of code for the return statement, which returned an error code. Thus, we can safely say that developers are generally not doing complex operations in the label blocks of their `goto` statements. This of course does not apply to the few outliers, in which the label blocks can grow very large. In particular, in cases of backwards jumps, we saw label blocks that spanned more than half the size of functions.

TABLE III: Results at the function level for the manual tagging of 384 randomly sampled functions for the purposes and properties of `goto`

Tage Name	% among All Files (384)	% among System Files (312)	% among Non-System Files (66)
Error	80.21	82.69	75.76
Cleanup	40.36	38.78	51.52
Control-Exit	10.16	9.94	12.12
Loop-Create	8.85	8.33	12.12
Spaghetti	5.99	6.09	6.06
Single	54.17	56.09	50
Multiple	62.24	63.46	62.12
Forward	90.1	92.31	87.88
Backward	14.06	13.46	18.18
Stacked labels	26.3	29.17	15.15
# statements in label block (median not %)	4	4	3

E. Results at the function level—Purpose

Tagging the files in the sample for classifying the purpose of `goto` statements, we found the following (see Table II for a definition of the purposes and Table III for the numerical results).

Most goto usage is for error handling: In manually tagging our dataset, we found that developers use `goto` statements for the purpose of error handling in $80.21 \pm 5\%$ of the functions. This can be explained in part because the C programming language does not have an explicit error handling mechanism like exception handling in C++/Java. Therefore, developers are using the combination of `goto` statements and the code block in the labels to emulate a `try/catch` mechanism (see code example below). However, note that exception handling with `goto` statements is a limited form of exception handling compared to that found in C++/Java: handlers are limited to be in the same procedure definition.

```
int fun (int x)
{
  code...
  if(error)
    goto err_label;
  code...
  err_label:
    print(error);
    cleanup(mem);
    return 0;
}
```

The second most frequent purpose is cleanup: Similar to the lack of exception handling capabilities, the C programming language does not have a cleanup construct such as `finally`, unlike C++/Java. Recall that a `finally` block in C++/Java is executed upon exiting a method, regardless of whether the exit is through a standard return or throw a raised exception. Therefore, in order to capture the cases where developers use `goto` statements to duplicate the purpose of `finally`, we tag a particular use of `goto` as cleanup, only if the code block in the label can be reached even when a `goto` statement is not executed (like in the example code above). We also found that developers use `goto` statements for the purpose of

PeerJ PrePrints | http://dx.doi.org/10.7287/peerj.preprints.826v1 | CC-BY 4.0 Open Access | rec: 11 Feb 2015, publ: 11 Feb 2015

PeerJ PrePrints

cleanup activities such as memory de-allocation at the end of a function, in $40.36 \pm 5\%$ of the functions.

Error handling and cleanup happened frequently together, but not always: We also found that cleanup occurred very frequently with error handling (like in the example code above). In $31.77 \pm 5\%$ on the functions, developers used `goto` statements for both error handling and cleanup activities. When both error handling and cleanup happens in the block of code under the label, the developers are essentially adding the code that is meant to be in the 'catch' block (error handling code), to the 'finally' block.

In the remaining $48.44 \pm 5\%$ of the times $(80.21 - 31.77)$, the developers used `goto` statements exclusively for error handling. In the cases where developers used `goto` statements exclusively for error handling, the code block in the label could be reached only if a `goto` statement was executed (see code example below). Such a behaviour is similar to having a 'catch' block, but no 'finally' block, in languages like C++/Java.

```
int fun (int x)
{
  code...
  if(error)
    goto err_label;
  code...
  return 1;
  /* because of the above return, the goto is not
       classified as cleanup, since the code block
       below is executed only if goto is executed. */
  err_label:
    print(error)
    return 0;
}
```

Alternatively in a small percentage of functions $(8.59 \pm 5\%)$, the developers use `goto` statements for cleanup, but not error handling. In such cases, the developers intend the execution to jump to a label where cleanup occurs (whether or not a `goto` statement was executed). However, the `goto` statement itself is not executed due to an error. The developers just wanted the function to end in certain flows.

Less intuitive usages such as control-exit and loop-create are less common: We found that developers used `goto` statements for exiting out of standard loops (`for`, `while`, `do-while`), and in some cases to create loops manually instead of using a standard repetition construct. In $10.16 \pm 5\%$ of the functions, developers used `goto` statements to exit out of a normal loop. This is in-spite of the C programming language having a construct to exit out of a loop. We found that developers used this either for skipping code outside the loop or the exit fully out of a nested loop.

In $8.85 \pm 5\%$ of the functions, developers used the combination of a `goto` statement and a label to create loops (see example code below). When a certain condition was satisfied, a `goto` statement was executed, which took the execution backward to a label that is physically above the `goto` statement in the function, in order to create a loop. This is perhaps, one of the least intuitive uses of `goto` statements, since looping constructs exist in C programming language. Only in the case of loop-create, did we not find a reason for the developers to use them. This pattern of usage was surprising to us, and it would be interesting in the future, to interview developers and see what advantages they found in manually crafting a loop based on `goto`.

```
int fun (int x)
{
  //loop create
  loop_create_label
    code...
    if(condition)
      goto loop_create_label;
  return 0;
}
```

Summary: We find that an overwhelming majority of the gotos are used mostly for two related purposes: error handling and cleanup. Thus as predicted by Tribble [13] (and not as emphasized by Knuth [12]), in the absence of dedicated constructs to handle these special operations (such as `try/catch` and `finally` in C++/Java), C programmers resort to using `goto` for error handling and cleanup, and do so in a rather disciplined way: most functions we surveyed had their error-handling or cleanup blocks systematically located at the end of the function. Only a minority of functions use `goto` statements for other purposes, such as breaking out of nested loop, or creating loops in an ad-hoc manner (which was not recommended by Knuth [12]).

F. Findings at the function level—Properties

In a majority of functions, multiple `goto` statements jump to the same label: In $62.24 \pm 5\%$ of the functions, developers have several instance of `goto` statements that jump to the same label. Such a use of `goto`, was one of the main arguments against the use of `goto` by Dijkstra. By having multiple `goto` statements jump to the same label, a developer cannot know which `goto` statement brought the execution to the code of the label block. Additionally in the cases where multiple developers are editing the same file, a developer cannot change the code in the label without knowing why each of the `goto` statements (several of which could have been introduced by other developers) are jumping to that label. However, the frequency of the use of `goto` statements in such a manner, could indicate that developers are comfortable with it.

Other jumping patterns are simpler to understand: On the other hand, $54.17 \pm 5\%$ of functions contained labels where each of the labels are associated with a single `goto`. Each of these functions, could have one or more labels, but each of them are associated with exactly one `goto` statement. Additionally, there exists a subset of the functions in which there is only one label block and it is unreachable by the normal execution, and only reachable by a single goto statement. In these functions, it is only possible to enter the label block via a single point of entry, and hence Dijkstra's main argument (not knowing from where the execution jumped to the label) does not apply. These functions constitute 8.85 ± 5 % of the functions we surveyed.

PeerJ PrePrints

PeerJ PrePrints | http://dx.doi.org/10.7287/peerj.preprints.826v1 | CC-BY 4.0 Open Access | rec: 11 Feb 2015, publ: 11 Feb 2015

Stacking labels at the bottom of functions is prevalent: We find that in $26.30 \pm 5\%$ of the functions, developers stack all the labels at the end of the function. An interesting pattern for error functions is labels that are stacked in the reverse order they are referenced in a function. Hence the farther in the function the failure occurs, the most actions to undo without needing to duplicate code. This provides a concise solution to undoing a series of steps that can individually fail. Some of the functions in our corpus had upwards of 4 labels stacked in such a way. Such a solution is not easy to accomplish with a try/catch block in C++/Java. Therefore, this is one of the cases where `goto` statements and appropriately placed label blocks might be simpler than a try/catch block.

Spaghetti code is uncommon: We found that in a few cases ($5.99 \pm 5\%$), the developers used `goto` to create spaghetti code – `goto` statements inside the code block that is under the label of a different `goto` statement. Such a use of `goto` allows arbitrary jumps within the code of the function, thus making it difficult to keep track of which line of code is currently being executed. Overall, such behaviour exemplifies the really harmful usages of `goto`, especially when used in complex functions. Nevertheless, we stress that we encountered a very small amount (6%) of such cases.

Most jumps are forward, not backwards: We found that in $90.55 \pm 5\%$ of the functions, developers used `goto` statements to jump to a label that is physically after the corresponding `goto` statement (like example code above). On the other hand, developers used `goto` statements to jump to a label that was physically before the corresponding `goto` statement in the code, in a relatively small, but non negligible, portion of the functions ($14.06 \pm 5\%$, note that the percentage of backward and forward jumps do not add up to 100% since there are functions where both such jumps exist).

As we saw before, developers used `goto` statements for creating loops. In order to create loops, the developers would have had to use a `goto` statement that jumped backwards to a label (see above for example to create loops). Unsurprisingly, most usages of backward `goto` statements are for this purpose. However, we found that in $5.21 \pm 5\%$ of the functions, developers jumped backwards to a label from a `goto` statement, but were not creating a loop (because the `goto` statement was not part of a condition). Every time the code executed such a `goto` statement, the execution would go back to a label.

Additionally we found that in $4.17 \pm 5\%$ of the functions, a backward jump from a `goto` statement was done from within the label of another `goto` statement, thus making it both a backward jump and spaghetti code. Such a use of `goto` might be very hard to keep track of during execution. That may be the reason why we find so few cases of this.

Summary: We find that the usages of `goto` statements exhibit several properties. First, we find that it is common to have several `goto` statements jumping to the same label: if this presents the issue highlighted by Dijkstra that it is not possible to know exactly the control flow that led to the label, it also means code duplication has been avoided (similar to calling an inner function – something standard in programming languages that support them). On the other hand, many labels are only reached by a single `goto` ($54.17 \pm 5\%$ of the functions). Further, 8.85 ± 5 % of functions have label blocks that are only reachable by a single `goto` statement; in those cases, Dijkstra's main concern does not apply. We also find that backwards jumps are often, but not always, correlated with loop creation `goto` statements. Finally, arbitrary jumps between `goto` statements ("spaghetti") are a relatively rare occurrence alleviating concerns about their impact.

> *We find that, in general, the use of* `goto` *is actually well disciplined. Most uses of* `goto` *statements are reasonably structured, filling the void of missing higher-level constructs found in other languages. There are of course usages that are unstructured as Dijkstra feared, but they are overall in the minority.*

VI. RQ2: Are `goto` statements involved in post-release bugs?

A. Motivation

Since Dijkstra's article claims that `goto` statements can have harmful effects, we wanted to empirically examine if `goto` statements did cause harm. However, a causal relation between a programming construct and harm is difficult to establish. Hence, we approximate harm as post-release bugs or bugs that are likely to affect the users of software and that could not be detected in development and testing. Then we examine if `goto` statements were removed/modified in any of the post-release bug fixes of a particular release of six OSS projects. In doing so, we assume that if a `goto` statement is removed/modified in a post-release bug fix, then it is more likely to be associated with that post-release bug (harmful).

B. Approach

We examine 180 days of post-release commit history for the six oss projects shown in Table I. During this period we extract the code that was removed and the code that was added in every bug fixing commit to the git repository. We determine if a commit was a bug fix commit or not by looking for keywords such as 'bug', 'fix', 'issue' [24]. Then for each bug fix commit, we look at the added and removed code to see if `goto` was removed from the code or modified (when the same `goto` statement exists both in the added and removed code).

In some cases, the developers might include in the commit comment or in the comments section of the bug in the issue tracking repository that a `goto` statement was the cause of the bug, and that was the reason why they removed/modified it, but we cannot be sure that they will make such a comment. The only sure way of knowing is by examining the code that was committed to the source code repository (which is what we do in our paper). Also note that we only focus on the `goto` statement and not the labels, since the removal/modification of a label will implicitly always require a change to the actual `goto` statement. We also do not examine if any change was

PeerJ PrePrints | http://dx.doi.org/10.7287/peerj.preprints.826v1 | CC-BY 4.0 Open Access | rec: 11 Feb 2015, publ: 11 Feb 2015

TABLE IV: The number of times a goto statement is removed/modified in the post-release phase of a project

Project	All Post Release Commits			Bug Fix Commits			Number of goto statements in the code
	# Commits	# goto statements removed	# goto statements modified	# Commits	# goto statements removed	# goto statements modified	
clamav-devel	87	0 (0 %)	1 (1.1 %)	25	0 (0 %)	0 (0 %)	758
ghostpdl	446	9 (2 %)	14 (3.1 %)	143	4 (2.8 %)	8 (5.6 %)	576
gimp	126	0 (0 %)	0 (0 %)	84	0 (0 %)	0 (0 %)	4,557
openldap	158	4 (2.5 %)	0 (0 %)	112	3 (2.7 %)	0 (0 %)	2,101
postgresql	97	1 (1 %)	2 (2.1 %)	24	0 (0 %)	0 (0 %)	815
VTK	1,012	1 (0.1 %)	0 (0 %)	145	0 (0 %)	0 (0 %)	4,579

made to the code within the block of code under a label, since that is the action taken once the goto is executed, and does not imply that a goto statement caused a bug. It only implies that the code executed after the the goto call has a bug. Hence, we focus only on the actual goto statement.

C. Results

The results of our analysis is presented in Table IV. We found that there are anywhere between 87 and 1,012 commits to the release we examined for the six OSS projects. Among the post-release commits, there were 25-145 bug fixing commits. However, only in two projects (ghostpdl, openldap) were goto statements removed in post-release bug fix commits. In the remaining four projects no goto statement was removed in a post-release bug fix commit. We found only one project (ghostpdl) where the goto statements were modified (either moved in the code or the target label was modified). In ghostpdl, eight of the 143 post-release bug fix commits had a goto statement that was modified.

We also looked at all post-release commits instead of just the bug fixing ones. We found that now four projects had goto removed, and three projects had goto modified. However, in two of those four projects (postgresql, VTK), only one goto statement was removed. In total, between 0 and 9 commits had goto statements removed, and between 0 and 14 commits had goto statements modified. However, the corresponding total number of commits ranged from 87 to 1,012. The highest percentage of post-release commits with goto statements modified was 3.1% — which is very small.

We place the above results in context in two ways. (1) We first examine the number of goto statements present in the respective release of the six OSS projects. If the number of goto statements is small, then obviously the number of goto statements removed/modified in the commits will be small. We found that the six projects had between 576 and 4,579 goto statements in the respective releases. Therefore, the lack of goto statements being removed/modified in bug fix commits is not due to the lack of goto statements in the code; (2) We also looked at another programming construct to place the results regarding goto statements in context, namely if statements. We found that between one and 29 if statements were removed, and between four and 59 if statements were modified in the bug fix commits. In five out of the six projects the ratio of goto statements removed/modified to all goto statements is not significantly different from the ratio of if statements modified/removed to all if statements. Only in GIMP that ratio is higher for goto statements and the difference is statistically significant. This means that only in one out of six projects post-release fixes were more likely to modify a goto statement than an if statement after adjusting for the relative frequency of goto and if statements. Therefore the number of goto statements removed/modified are much smaller in comparison.

Finally, we manually analyze the post-release bug fix commits in which goto was removed/modified. The two projects with such commits are ghostpdl and openldap. *In none of the commits were goto statements removed/modified because they caused a post-release bug*. There were three reasons for a goto to be removed/modified: (1) A feature was removed and the goto within the code of the feature was no longer needed; (2) goto was moved due to code/comments added somewhere above the goto statement; and (3) goto was moved to a different if-condition's block of code (only one case). We noticed that no goto was modified in such a way that it jumped to a different label.

Additionally, we found that all but two of the goto statements in the commits were used for cleanup or error handling purposes. Among the two remaining goto related commits: one was a goto that was modified (similar to case 2 above) in ghostpdl. It was used for spaghetti code; another was a goto that was removed (similar to case 1 above) in openldap. It was used for a loop exit. Interestingly, the loop that it was exiting out of was created with a goto as well. Also along with the goto the label was removed as well.

> *If we assume bugs in the post-release phase of a project as a measure of harm, then the small number of goto statements being removed/modified in bug fixes implies that goto statements were not considered harmful enough to be removed/modified in the post-release phase of the project in most cases.*

VII. Threats to Validity

False positives due to code generation: the heuristic we used may not be sufficient to filter out all generated files. However, the manual inspection of the random sample provides us with a very small estimate of generated files that are left in the sample ($1.5 \pm 5\%$).

PeerJ PrePrints | http://dx.doi.org/10.7287/peerj.preprints.826v1 | CC-BY 4.0 Open Access | rec: 11 Feb 2015, publ: 11 Feb 2015

False positives due to other reasons: the regular expression we used to find gotos may return false positives, however our manual analysis uncovered only one such case (a goto in commented-out code).

Overestimation due to forks: projects in Github are easy to fork, and that may lead to a potential proliferation of very similar projects in the population. However, none of the files in the random sample of 384 files we manually examined in RQ1 were from forked projects.

Sampling errors: we performed a simple random sampling without replacement at the file level to get the list of files to inspect. We then followed that by sampling one function per file for detailed inspection. An alternative was to sample directly at the function level which conflicted with our goal to perform the analysis at the level of both files and functions. Another alternative was to use all the functions in the file, which may bias the results towards files with large number of functions with `goto` statements. As such there is a small threat the sample is not representative. However, our way of sampling is similar to the way Gallup samples people for its polls [23]. Yet another alternative to sampling is to use an automated tool like Coccinelle [25] to automatically determine patterns. However, we choose not to use an automatic parser for three reasons: (a) we could have identified several properties like forward or backward jumps, but it would have been more difficult to identify spaghetti code; (b) manually analyzing a random sample should give an accurate representation of the whole population; and (c) we had to do a manual inspection to determine all the different uses of `goto`.

Goto semantics specific to C: C's goto can only jump inside a function. Other languages may permit other things, and have different usage patterns, as such further studies are needed in those languages. We also searched for setjmp, C's equivalent to a goto that can jump to other functions, but found very few usages of it.

Generalizability: We conducted our study on the set of C source code projects available on Github, which is the largest open-source repository in use at the moment. As such, we expect that the results of this study are somewhat generalizable, unless the C projects available on github are markedly different from other C projects in existence.

Domain of the files: there may be different domains for files using `goto` statements, compared to other C files. We found that $85 \pm 5\%$ of the files with `goto` statements in the dataset that we inspected manually are system or networking files. In terms of type of usage, we found that the proportions of `goto` statements defined for a certain purpose or exhibiting a certain property were comparable between system and non-system files, with the notable exception of the "stacked labels" property. If we cannot discard the possibility that there are differences in usage in terms of domains, we have not encountered strong evidence towards this either. In any case, further studies on a larger sample of non-system files need to be conducted.

VIII. Conclusion

For decades, there have been arguments over the use of `goto` in programs. However, these arguments have been mostly theoretical. This study is to our knowledge the first to bring empirical evidence to the debate; by reporting on how C developers are actually using `goto` in practice, and by investigating if they are involved in post-release bugs. We conducted a large-scale study of more than 11K projects (more than two millions C files), featuring both quantitative and qualitative components. Summarizing our findings:

- **Most usages of `goto` statements appear to be reasonable.** Our qualitative study of a sample of files and functions using `goto` shed light on why `goto` statements are in use: far from being maintenance nightmares, most usages of `goto` follow disciplined, well-designed usage patterns, that are handled by specific constructs in more modern languages. The most common pattern are error-handling and cleanup, for which exception handling constructs exist in most modern languages, but not in C. Even properties such as several `goto` statements jumping to the same label have benefits, such as reducing code duplication, in spite of the coordinate problem described by Dijkstra.
- **Developers did not remove/modify `goto` statements in the post-release phase of four of the six projects.** Finally, our quantitative study on the post-release history of six OSS projects showed that developers did not find `goto` statements harmful enough to be removed in a bug fix commit in four of the six projects. Even in the two projects where `goto` statements are removed/modified, we notice through manual inspection that `goto` statements did not cause the bugs. Thus we do not find evidence that `goto` statements caused post-release bugs in these projects.

Overall, our qualitative study tells us that only a minority of `goto` usages are really exemplary of the 'disastrous effects' Dijkstra warned us about. We have no evidence that `goto` statements were used differently prior to Dijkstra's famous article than now. However, the current usage heeds Dijkstra's advice, but not to the letter: `goto` is being used, but is mostly limited to the cases where it actually offers an improvement over the alternatives. Indeed, some developers perceive that sometimes `goto` is the cleanest way to achieve something [26]. Thus `goto` statements do not appear to cause harm in practice, as they are mostly used for a limited set of straightforward purposes, quite unlike the ways Dijkstra feared they may be misused.

Our investigation raised several questions. In particular, more work is needed to determine if there are systematic differences between the system code and the non-system C code (though some may argue that C is designed primarily for system code). Surveys and interviews with developers will help corroborate and enrich the findings presented here.

References

[1] E. W. Dijkstra, "Letters to the editor: Go to statement considered harmful," *Communications of ACM*, vol. 11, no. 3, pp. 147–148, Mar. 1968.

PeerJ PrePrints | http://dx.doi.org/10.7287/peerj.preprints.826v1 | CC-BY 4.0 Open Access | rec: 11 Feb 2015, publ: 11 Feb 2015

PeerJ PrePrints

[2] Username: MaD70, "Goto still considered harmful?" http://stackoverflow.com/questions/46586/goto-still-considered-harmful, Aug 2012.
[3] W. Wulf and M. Shaw, "Global variable considered harmful," *SIGPLAN Not.*, vol. 8, no. 2, pp. 28–34, Feb. 1973.
[4] C. Ponder and B. Bush, "Polymorphism considered harmful," *SIGPLAN Not.*, vol. 27, no. 6, pp. 76–79, Jun. 1992.
[5] C. A. Kent and J. C. Mogul, "Fragmentation considered harmful," *SIGCOMM Comput. Commun. Rev.*, vol. 25, no. 1, pp. 75–87, Jan. 1995.
[6] Eric A. Meyer, ""considered harmful" essays considered harmful," http://meyerweb.com/eric/comment/chech.html, Dec 2002.
[7] L. Marshall and J. Webber, "Gotos considered harmful and other programmers taboos," 2000.
[8] Andrew Koenig, "What dijkstra said was harmful about goto statements," http://www.drdobbs.com/cpp/what-dijkstra-said-was-harmful-about-got/228700940, Mar 2009.
[9] S. Saha, J. Lawall, and G. Muller, "An approach to improving the structure of error-handling code in the linux kernel," in *Proceedings of the 2011 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, ser. LCTES '11, 2011, pp. 41–50.
[10] S. Saha, J.-P. Lozi, G. Thomas, J. Lawall, and G. Muller, "Hector: Detecting resource-release omission faults in error-handling code for systems software," in *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, June 2013, pp. 1–12.
[11] "Github," https://github.com/, Checked Aug 2014.
[12] D. E. Knuth, "Structured programming with go to statements," *ACM Computing Surveys*, vol. 6, no. 4, pp. 261–301, Dec. 1974.
[13] Edsger W. Dijkstra, "Go to statement considered harmful: A retrospective," http://david.tribble.com/text/goto.html, Nov 2005.
[14] ——, "What led to "notes on structured programming"," http://www.cs.utexas.edu/ EWD/transcriptions/EWD13xx/EWD1308.html, Jun 2001.
[15] Robert Love, Rik van Riel, Linus Torvalds, "Linux: Using goto in kernel code (mailing list discussion)," http://goo.gl/ItZHiu, Jan 2003.
[16] A. Mockus, "Amassing and indexing a large sample of version control systems: Towards the census of public source code history," in *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, ser. MSR '09, 2009, pp. 11–20.
[17] M. Nagappan, T. Zimmermann, and C. Bird, "Diversity in software engineering research," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, 2013, pp. 466–476.
[18] E. Shihab, "An exploration of challenges limiting pragmatic software defect prediction," Ph.D. dissertation, School of Computing, Faculty of Arts and Science, Queen's University, 2012.
[19] "Download statistics for GIMP in Sourceforge," http://goo.gl/AiUg8f, Checked Aug 2014.
[20] D. P. Duncavage, "Nasa needs postgres - nagios help," http://goo.gl/AyY8J2, Checked Aug 2014.
[21] M. Krieger, "Keeping instagram up with over a million new users in twelve hours," http://goo.gl/415Zrk, Checked Aug 2014.
[22] "Sample size calculator," http://www.surveysystem.com/sscalc.htm, Checked Aug 2014.
[23] "How does gallup polling work?" http://www.gallup.com/poll/101872/how-does-gallup-polling-work.aspx, Checked Aug 2014.
[24] A. Mockus and L. Votta, "Identifying reasons for software changes using historic databases," in *Software Maintenance, 2000. Proceedings. International Conference on*, 2000, pp. 120–130.
[25] "Coccinelle," http://coccinelle.lip6.fr/, Checked Aug 2014.
[26] M. Szeredi, "Commit - fuse: fuse_fill_super error handling cleanup," http://goo.gl/kRPJva, Checked Aug 2014.

PeerJ PrePrints

PeerJ PrePrints | http://dx.doi.org/10.7287/peerj.preprints.826v1 | CC-BY 4.0 Open Access | rec: 11 Feb 2015, publ: 11 Feb 2015