Change Impact Graphs: Determining the Impact of Prior Code Changes

Daniel M. German*

dmg@uvic.ca
Dept. of Computer Science
University of Victoria, Canada

Ahmed E. Hassan

ahmed@cs.queensu.ca
Queen's University, Canada

Gregorio Robles

gregorio.robles@urjc.es
Universidad Rey Juan Carlos, Spain

Abstract

The source code of a software system is in constant change. The impact of these changes spreads out across the software system and may lead to the sudden manifestation of failures in unchanged parts. To help developers fix such failures, we propose a method that, in a pre-processing stage, analyzes prior code changes to determine what functions have been modified. Next, given a particular period of time in the past, the functions changed during that period are propagated throughout the rest of the system using the dependence graph of the system. This information is visualized using Change Impact Graphs (CIGs). Through a case study based on the Apache Web Server, we demonstrate the benefit of using CIGs to investigate several real defects.

1. Introduction

All too often when maintaining a large software system, a bug report is submitted regarding changes in the behavior of an unchanged functionality. Investigating this type of bug reports is difficult and tedious, since the fix is frequently in a different location than where the failure manifests itself, i.e., the location specified in the bug report. The failing behavior is usually due to the ripple effect of another change in a different part of the system that propagates along various dependencies, such as call and data dependencies, and affects the unchanged code.

The maintainer in charge of fixing such failures starts her investigation with the location where the failure manifests itself. She then examines the dependency graph of the reported failing function in an ad-hoc manner using her knowledge and her experience about the software system trying to pin-down the actual location of the bug causing the failure. A maintainer could use slicing techniques [1, 2] to determine all the code locations which may affect the reported location of a failure and are likely the source of the bug causing the failure. However, slicing techniques are known to report large slices [3, 4] and are often limited to small to medium software systems [5]. A single slice may contain as much as as 30% of the source code of an application. Maintainers would spend considerable time investigating such large slices for complex real-life software systems. Approaches, such as dynamic slicing [6, 7], have been proposed in literature to reduce the size of slices and make them more accurate for large software systems. However most techniques require additional effort (e.g., execution of tests for dynamic slicing) and expensive analyses.

In this paper, we propose a method which determines the impact of historical code changes on a particular code segment (e.g., a function). Given the reported location of a failure, a maintainer wants to know of any recent code changes which could have impacted the functionality of the failing function—specially if that function was not changed recently. Our method determines all the changed areas of the software system which affect the reported location of a failure. The method then annotates these parts by marking recent code changes and propagating the impact of these recent changes. It then creates a *change impact graph* to determine what areas might have been affected by certain changes to help maintainers rapidly pinpoint the source of a bug given the reported location of a failure. The maintainer needs to only examine the marked up functions instead of going through all the functions which would be produced by a slicing technique. Our method should be seen as a time saving complement to slicing techniques, as it is a way of filtering slices.

We demonstrate the feasibility and possibilities of our method through an exploratory case study based on several real bug reports from the Apache Web Server. Through the bug reports, we demonstrate the benefit of using our method to investigate the reported failures and fix the corresponding bugs by non-experts.

*Corresponding author

Organization of the Paper

The remainder of the paper is organized as follows: the next section introduces our model for tracking the impact of historical code changes. Section 3 presents a methodology to analyze historical code changes and recover their impact on source code entities (i.e., functions). Section 4 presents an exploratory case study in which we use this methodology to pinpoint the changes that created four real bugs from the Apache Web Server. Section 6 discusses the effectiveness, limitations, and possible improvements for our method. Section 7 concludes the paper.

2. A model to track the impact of historical code changes

Historical changes to a function can be modeled as a sequence, where each element corresponds to the source code of the function after each particular change. Formally, for a function f we define its change history sequence as $H_f = \langle f_0, ..., f_m \rangle$, where f_i is the ith instance of the function. Each instance of a function, can be annotated with metadata about the change such as its data, its purpose and the name of the developer who performed the change.

The dependence graph of a function f, $G(f)$, is modeled as a directed graph. Its nodes are the functions that are reachable from f and its edges are the direct calls between any of these functions. If a function g is called from function f, the dependence graph of f contains the dependence graph of g. The dependence graph can be considered a simplified interprocedural dependence graph that only tracks function invocation and does not track in or out parameters nor variables. The dependence graph of f includes any function that could be called by f. When a developer peruses the source code of a function f, she is not usually aware of all the contents (or the size) of this dependence graph. She is only aware of the edges that start in f (the function calls inside f).

The dependence graph of a function f can be created at any time t (during the life of such a function). The graph is built recursively as described above, using the latest instance of every one of all the functions in the graph such that their date of modification is less or equal to t. In other words, if we want to build the dependency graph of f on Dec. 31, 2007, then we will use the latest instance of f with a date less or equal to Dec. 31, 2007. If it calls a function g then we will use the latest instance of g with a date less or equal to Dec. 31, 2007. This process continues until the dependence graph is completed.

The dependence graph of a software system is the union of the dependence graphs of all its functions.

We illustrate our model with a simple example. Assume a C source file that has had four changes recorded as depicted in Figure 1. The change history for its functions is shown in Figure 2. The change history tracks when the functions are added, deleted or modified.

2.1. Propagation of prior changes

A typical use-case involves a developer who is perusing the source code of function f at time t, and who is interested to know any changes that might have had an impact on the behavior of f during a particular time window $[t_b, t_e]$ in the past. We

C_0	C_1	C_2	C_3

`void a() {`
`b();`
`c();`
`d();`
`}`
	`void a() {`
	`b();`
	`c();`
	`d();`
	`}`
		`void a() {`
		`b();`
		`c();`
		`d();`
		`}`
			`void a() {`
			`b();`
			`c();`
			`d();`
			`}`

`void b() {`
`e();`
`}`
	`void b() {`
	`e();`
	`}`
		`void b() {`
		`e();`
		`}`
			`void b() {`
			`e();`
			`}`

`void c() {`
`var2=1;`
`}`
	`void c() {`
	`var2=1;`
	`}`
		`void c() {`
		`var2=1;`
		`}`
			`void c() {`
			`var1=5;`
			`}`

`int d() {`
`return 0;`
`}`
	`int d() {`
	`exit(1);`
	`}`
		`int d() {`
		`exit(1);`
		`}`
			`int d() {`
			`exit(1);`
			`}`

`int e() {`
`f();`
`}`
	`int e() {`
	`f();`
	`}`
		`int e() {`
		`return 0;`
		`}`
			`int e() {`
			`return 0;`
			`}`

`int f() {`
`var1=0;`
`}`
	`int f() {`
	`var1=0;`
	`}`		

Figure 1: Evolution of the source code of an example system after four different changes ($C_0, ...C_3$). The areas affected by each change are shown in bold.

	C_0	C_1	C_2	C_3
a	A			
b	A			
c	A			M
d	A	M		
e	A		M	
f	A		D	

Figure 2: Depiction of the change history for the functions of the example system. The rows correspond to the functions and the columns to the changes. A, M, D are, respectively, *Added*, *Modified* and *Deleted*. `exit` is not included because it is a C library function, external to the system being maintained.

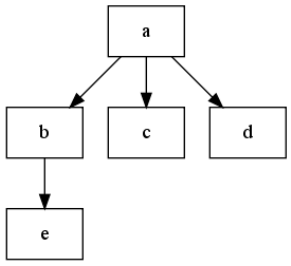

Figure 3: Dependence graph of $a()$ immediately after change C_3.

call this time window the period of interest. The period of interest does not need to include changes up to time t. For example, the graph can be created in December using a period of interest that spans the previous April to May.

To determine the impact of prior changes on a particular function f, the dependence graph of f is computed at time t and its nodes are marked according to any changes during the period of interest $[t_b, t_e]$ as follows:

1. Mark all nodes in $G(f)$ as *unaffected*.
2. For each node g in $G(f)$: if it has been added or changed during $[t_b, t_e]$, then annotate it as *changed*.
3. Repeat until the dependence graph, being built, stops changing:
   - for any node that is still *unaffected*, mark it as *affected* if at least one of its successors is either *changed* or *affected*.

CIGs can have cycles (the result of recursive calls). The algorithm is guaranteed to terminate because it processes, for every pass, each node once; and for each node it would need to check at most each of its successors. The number of passes will also be finite (at most equal to the number of nodes in the graph).

Each node in the resulting dependence graph, which we call a *Change Impact Graph* or CIG, is one of three types:

1. *Unaffected*. The function nor any of the functions it can potentially call were affected by the changes.
2. *Changed*. The source code of the function has been changed.
3. *Affected*. The source code of the function has not changed, but at least one of the functions it can potentially call has changed.

Figure 4 shows the CIG for the example of Figure 1. The GIC has been computed using the callgraph after change C_3, and its period of interest includes the changes C_1 and C_2. The change to $e()$ is propagated to $b()$, and then to $a()$ (which is also affected by the change to $d()$). The graphs shows that the functionality of both $a()$ and $b()$ might have been affected by these changes, but not the functionality of $c()$.

2.2. Pruning CIGs

A typical problem of dependence graphs (and as consequence CIGs) is that they may contain too many nodes. We propose the following methods to prune them. Our goal is to remove nodes that are not of interest.

2.2.1. Remove unaffected nodes

Once the CIG has been computed, we remove nodes that are not affected. By reducing the number of nodes the rendering of the CIG is typically simplified. Figure 5 shows our sample CIG after it has been pruned in this manner.

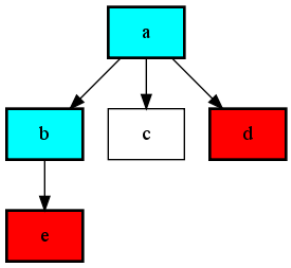

Figure 4: Change Impact Graph of $a()$ computed using the source code immediately after C_3 but only showing the propagation of the changes C_1 and C_2 (the period of interest includes only these two changes). Red depicts *changed* functions, light blue corresponds to *affected* functions, and white to *unaffected* (*changed* will appear darker than *affected* in black-and-white versions of these images).

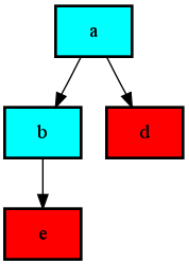

Figure 5: The CIG of Figure 4 with only affected and changed nodes.

2.2.2. Remove nodes outside the area of interest

Frequently a developer is only interested in a specific area (e.g., a particular subsystem) of the codebase, and would like to know only when this area has been changed, and affected by changes outside it (this provides awareness that some change outside this area might have affected the functionality in question).

In other scenario the developer is certain that the defect is located in a specific area. In such case she is not interested in knowing any changes outside it, nor any functions that might be affected by them.

To address these two issues we propose two variants to the pruning of nodes outside an area of interest:

Prune-Before. The dependence graph is pruned before the impact of the changes is computed. This method can be summarized as follows: We start by computing the dependence graph then removing the nodes outside the area of interest. We compute the CIG using the resulting dependence graph. The resulting CIG does not show the effect of changes to areas outside the area of interest. A Prune-Before CIG is exemplified in Figure 6. In this case functions $a()$, $b()$, $c()$, and $d()$ are the area of interest; note how the impact of the change to $e()$ is no longer depicted in the CIG.

Prune-After. The CIG is pruned to remove nodes outside the area of interest, yet the impact of such nodes will still be depicted in the GIC. The method to prune the CIG is simple: We start by computing the CIG, then we remove the nodes outside the area of interest. A Pruned-After

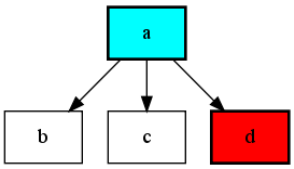

Figure 6: The result of applying "prune-before" to the CIG of Figure 4. In this example we assume that $a()$, $b()$, $c()$ and $d()$ are the area of interest; e is removed before the CIG is computed. The resulting CIG does not depict the impact of the change to e.

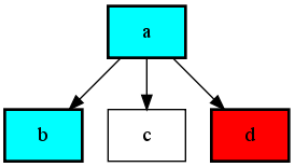

Figure 7: The result of applying "prune-after" to the CIG of Figure 4. Like Figure 6, $a()$, $b()$, $c()$ and $d()$ are the area of interest. Nodes outside the region of interest (in this case e) are removed after the CIG has been computed. The effect of the change to $e()$ is still shown in $b()$–which appears as affected.

CIG is exemplified in Figure 5. Like Figure 7 $a()$, $b()$, $c()$ and $d()$ are the area of interest. In this case the change to e is propagated to its caller $b()$ before $e()$ is removed from the graph. This CIG shows that $b()$ has been affected by a change (even if we do not know what prompted such change).

Both prune-before and prune-after CIGs can have their unaffected nodes removed, resulting in a CIG that shows only changed and affected nodes within the area of interest.

2.3. Annotating CIGs

The nodes of the CIGs can be annotated using visual attributes, such as size, colour, shape, and textual information. These visual attributes are used to highlight specific properties of the changed and affected nodes. For example, the brightness of the node can show how old the change is with paler nodes representing newer nodes, and brighter nodes representing more recent ones. The size of the node can correspond to the number of times it has been changed, or different colours can be used to depict changes by different developers.

2.4. Quantifying the impact of changes

We define two metrics to quantify the effect of the changes during a period of interest: the *ratio of changed functions* and the *ratio of affected functions* in the CIG of a function.

- The *ratio of affected functions* is the proportion of *changed* and *affected* to the total nodes in a CIG (of a function or a system). It provides an overview of the area impacted by the changes. If a set of changes have a large ratio of affected functions, then such changes have the potential to affect the functionality of a large proportion of the functions in the software system. Using our running example shown in Figure 4, the ratio of affected functions is 4/5.
- The *ratio of changed functions* is the proportion of *changed* nodes to the total nodes in a dependence graph. While the ratio of affected functions gives the impact that changes have in the software, the ratio of changed functions provides the number of functions that may be the origin of the bug. This ratio gives an overview of the proportion of changed functions. Using our running example shown in Figure 4, the ratio of changed functions is 2/5.

In practice, the higher the ratio of affected functions is, the more areas a failure-inducing change could affect. By computing the ratio of affected functions of a potential change a developer could assess the criticality of a change.

When a developer computes a CIG, she will want to minimize the ratio of affected and changed functions. She will usually work with the current version of the source code, and specify a period of interest in the past. She will want to narrow potential areas of the code that would have been affected during such changes. The longer the historical period of interest, the higher the ratio of changed functions, making this method less effective. The major challenge when using a CIG is finding a suitable period of interest such that the buggy change which introduced the failure (or any other interesting functionality) is within it, while minimizing the ratio of nodes (i.e., nodes) in the CIG.

2.5. Annotating Source Code

Dependence graphs of real systems are usually complex and difficult to read or visualize. We propose instead to annotate the source code of any function with the help of CIGs. In its most simple conception, each line of code will be tagged if it contains a call to a function that is marked *affected* or *changed*. We refer to this source code view as the *impact-annotated source code*). Figure 8 shows the impact-annotated source code of our running example after change C_3, with the period of interest comprising of changes C_1 and C_2. Using its corresponding CIG (as depicted in Figure 4), the calls from $a()$ to $b()$ and $d()$, and from $b()$ to $e()$, have been coloured as affected; two statements are coloured as changed. The colour scheme is the same as the one used in the CIG: *affected* statements are shown in light gray (Gray 71), and *changed* ones in dark gray (Gray 41). The colouring of the source code gives awareness to the developer of what was affected during the period of interest.

Let us assume that a failure was reported in $a()$ after C_2, and that this failure did not exist before C_1. In other words, the failure is presumed to have been caused by a bug introduced during changes C_1 or C_2 (or both). The developer will probably start by inspecting function $a()$. The call $c()$ is not likely to be the cause of the failure (it is not changed nor affected) and could be ignored (or at least presumed to have a lower probability of being the location of the bug). On the other hand, calls $b()$ and $d()$ are marked as affected, so it is worth exploring both function $b()$ and $d()$ to see if the change to one of them (or its successors) has introduced the bug. The goal of impact-annotated source code is to guide the attention of the developer towards the functions that are more likely to be responsible for a failure.

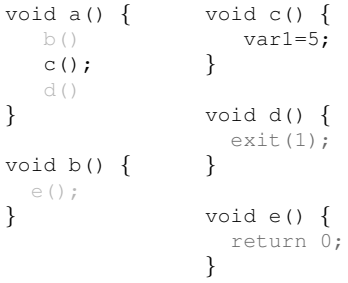

Figure 8: Impact-annotated source code for our example system after change $C3$ for a period of interest that includes changes C_1 and C_2. Dark gray statements were changed during this period, and light gray ones were affected by these changes. It can be seen that a was not changed during this period, but its calls to b and d were affected because of the changes to d and e; meanwhile c was not affected by changes during this period.

3. Recovering the impact of function evolution from a version control system

In this section we present the implementation of the model described in Section 2. We assume that the source code history is stored in a version control system (such as `subversion` or `CVS`). Although we discuss our implementation within the scope of C, it is applicable to other (procedural) programming languages.

3.1. Recovering the change histories of functions

We use the information recorded in the version control system to compute the history of each function. Since version control systems track the evolution of a software system at the line level, we must perform additional analysis and extraction to recover the history of code changes at the function and dependency levels. This process is illustrated in Figure 9.

From the version control system, we process the code after each commit. We refer to a commit as a modification record–MR. For each MR we use the tokenizer of `ccfinder` [8] to compute and store a callgraph of the system. We then determine the functions affected by this MR as follows:

Each MR consists of changes to zero or more source code files, and results in a new instance (or version) for each of such files. For each instance of each file in the MR we perform the following operations;

Remove whitespace, comments and reformat. Sometimes a change affects only whitespace or comments, and occasionally it might affect a large number of functions. For example, PostgreSQL reformats its source code on a regular basis [9]. We want to skip these edits because these operations do not change the functionality of the code for most programming languages except for some programming languages such as Python which are indentation dependent. This processing ensures that regular reformatting of the source code would not result in many false positives with the reformatting operation appearing as changed functionality for those functions that have been reformatted. The same holds for comments so we ignore changes to comments as well.

Identify each function in the file. We use *exuberant ctags*[1] to identify the location where the definition of a function starts. The end of a function is assumed to be the location of the last closing brace before the next definition in the file. When processing C source code we do not consider macros as a definition, as macros can appear in the middle of a C function.

Determine the type of operation on the function. We compare each function against its previous instance. Each instance of a function is tagged as either: *unchanged* (it wasn't altered), *modified* (it has changed), *added* (it did not exist in the previous instance of the file).

Determine deleted functions. We tag any function that appears in the previous instance of a file, and not in the new instance of the file, as *deleted* during this MR.

We identify each function instance uniquely by its name, the filename where it is found, and the MR id (according to the version control system). This approach permits us to deal with multiply-defined functions such as C `static` functions.

One major challenge is the detection of functions that have been moved and/or refactored. It is interesting and valuable to have a precise picture of the history of a function, but this analysis is not required for our method. Our main goal is to provide awareness of changes, i.e., to know that the dependence graph of a function **has** changed, not necessarily **how** it has changed. Our method could be extended using one of several methods to recover renaming and refactoring, such as the ones described in [10, 11, 12]. We discuss this issue further in section 6.4.

We store in a historical database, the history of each function and metadata about each MR such as the list of changed files and the name of the developer who performed the changes.

In summary, at this point we have retrieved and stored the change history of each function ever present in the history of the system: when it is added, when it is modified, and when it is deleted. Information on the developers who have performed these actions can be easily obtained and be stored as well, although we do not consider it in our method. We have also computed a callgraph of the system after each commit (MR) to the version control system.

3.2. Creating the Change Impact Graph (CIG)

The process used to create a CIG is described in Figure 10. In our method, when a developer is looking for a source of a defect she knows (for instance, from a bug report) the function where it appears, knows when the defect appeared (time b), and has some idea of when the defect might have been introduced (time a). The developer creates the CIG by providing three parameters: the function of interest $f()$ (the root of the CIG), the time t when the dependence should be computed (usually the present), and a period of interest $[a, b]$ (the CIG will show the impact of changes after time a but before time b). The method to create the CIG is as follows:

[1] `http://ctags.sourceforge.net`

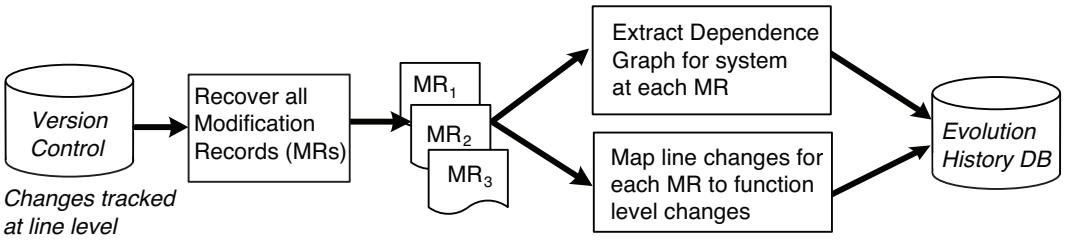

Figure 9: Process used to create the Evolution History Database.

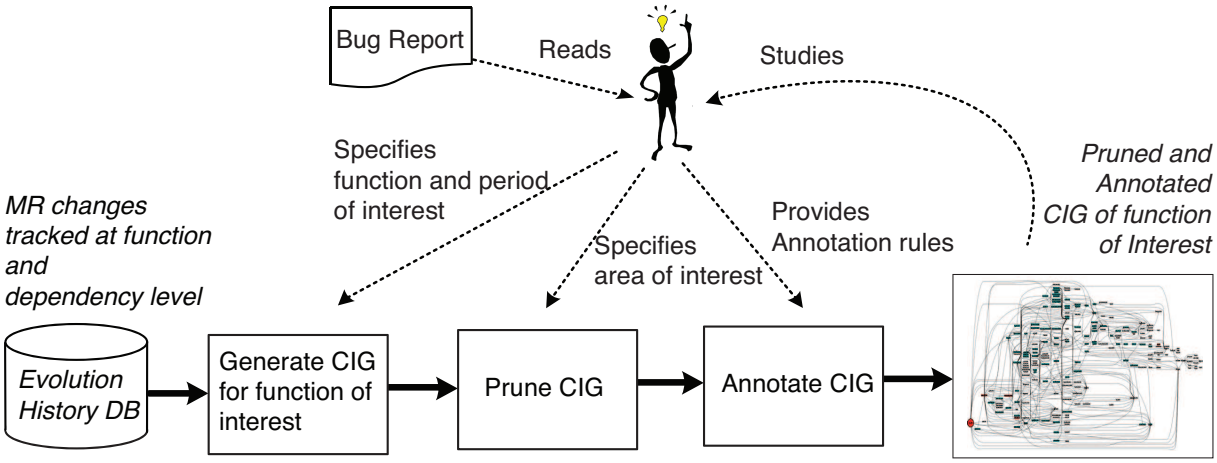

Figure 10: Creating and using a CIG to fix a bug report.

1. Retrieve the dependence graph of the system at time t, and from it compute the dependence graph of $f()$.
2. If necessary, prune-before the dependence graph to remove nodes outside an area of interest (e.g. outside a particular set of files).
3. Mark and propagate changes to functions in the dependence graph of $f()$ during period $[a, b]$ using the algorithms described in Section 2. The result is the desired CIG.
4. If necessary, prune-after the CIG to remove nodes outside an area of interest.
5. If necessary, annotate the CIG. This will facilitate the exploration of the CIG by marking nodes according to the developer who made the changes.

We envision these four steps as iterative as a developer experiments with different periods of interest, and with different pruning and annotation methods until she finds the source of the defect.

3.3. A method to create CIGs to fix defects

The creation of a CIG requires four parameters: the root of the CIG, the date at which it is created, the period of interest and, optionally, a pruning area. The effectiveness of CIGs will depend on the good selection of them. The following are guidelines to for their selection:

1. *Find root of CGI.* This corresponds to the function that is exhibiting the error.
2. *Determine date at which the CIG is created.* This is usually the present. We believe developers are mostly interested in inspecting current code (and its dependence graph).
3. *Determine a pruning area.* Determine, if possible, a likely area where the bug might exist. Knowing such an area relies on many factors, such as the description of the defect, and the experience and intuition of the developer.
4. *Determine a period of interest.* Sometimes a defect report provides information that allows us to narrow its appearance to the days in which the defect was introduced. This narrowing down process depends largely on the experience and knowledge of the developer. A tool that can be used to assist in the narrowing down process is the *Change Impact Overview*. This is a graph that plots the accumulated number of changed functions per day, given a starting date, as shown in Figure 11. In other words, by choosing one date (usually the upper end of the period of interest) one can observe the days in which functions are—and are not—changed.

In practice identifying the location of a defect is an iterative process. A developer will choose a starting point for the creation of the first CGI, and change parameters as she narrows down the source of the defect.

4. Case Study

We performed a case study to investigate the possibilities and limitations of our method. For our study, we used the Apache Web Server version 1.3. We selected Apache for several reasons: it is a large, complex and well-known software

system with a rich history and a large number of developers. In addition, its defect tracking database and version control system are publicly available.

Although version 1.3 is currently in its maintenance phase, it is still widely in use. It has approximately 86 kSLOCs and is mostly written in C. It has 8,021 commits (with 29,999 file revisions). More information about Apache, its community and its way of development can be found in [13]. We made a copy of its `subversion` repository to avoid overloading Apache's servers.

To demonstrate the usefulness of our method, we needed to identify historical code changes that resulted in the manifestation of a failure in a different area of the software system. We searched the source control system for description of changes (the commit logs) which included the words "introduced", "bug" and "PR" followed by a number. Changes that fix a bug in Apache usually include a reference to the bug in the defect system using the following syntax: PR #<number>. We located seven such changes. We selected the four most recent changes. These changes fixed the following bug reports: PRs #1352, #3130, #5389, #10090 and #10185. These reports are depicted in Table 1.

PR #1352. This defect was reported in the `cgi` module on Nov. 3, 1997. The person reporting the bug claimed that recent changes in the log system had introduced it. We can summarize the four main steps of our method as follows:

1. *Find root of CGI.* We use `cgi_handler()`, the entry point of the `cgi` module, where the bug is being reported.
2. *Determine date at which the CIG is created.* We use Nov 3, 1997, the day in which the defect was reported.
3. *Determine a pruning area.* If we assume that this change is expected to be in the source code of the module—the file `mod_cgi.c`—we can prune the CIG to include only functions in this file.
4. *Determine a period of interest.* This defect indicated "that recent error log changes introduced a bug". The Change Impact Overview of `cgi_handler()` is presented in figure 11. The number of changed functions in the complete CIG grows steadily over time. When the graph is pruned the growth is significantly slower. Looking at the Change Impact Overview, we note that the most recent change that affects `cgi_handler()` occurred on day 27 (2 functions), then the impact of the change grows again in day 97 (to 3 functions). If we assume that this change is expected to be inside `mod_cgi.c` then the defect could not have been introduced during the last 27 days. Hence, our starting period of interest will be changes made during the last 30 days.

Figure 12 shows the CIG of `cgi_handler()`, computed on Nov. 3, 1997, and including the changes during the last 30 days. As it can be seen, there have been a significant number of changes on the entire CIG of `cgi_handler()` during this period. Figure 13 shows the corresponding pruned-after CIG. Only 2 nodes were changed. Examining the two nodes, we find that one of them was the source of the defect (the function `log_scripterror()`).

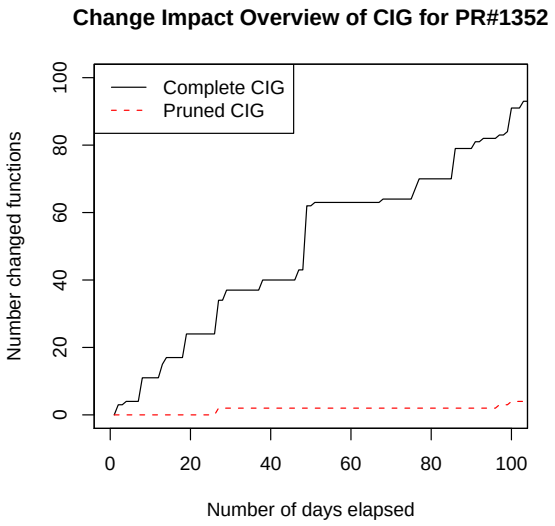

Figure 11: Change Impact Overview of `cgi_handler()` before Nov. 3, 1997. The number of changed functions that impact `cgi_handler()` grows steadily but the pruned CIG shows a very slow growth.

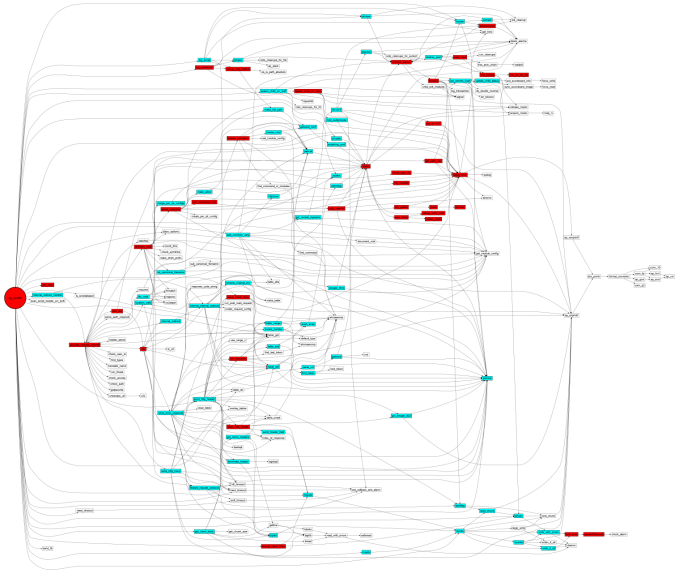

Figure 12: CIG of `cgi_handler` on Nov. 3, 1997, showing the propagated changes during the last 30 days.

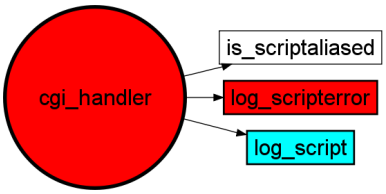

Figure 13: CIG of `cgi_handler` pruned-after, showing only changes inside `mod_cgi.c` on Nov. 3, 1997, representing the propagated changes during the last 30 days.

Problem Report	Date-Reported	Category	Main description
#1352	Nov 8 1997	mod_cgi	A coding issue in the mod_cgi.c module prevents the proper display in the error log file of the filename causing a specific error.
#3130	Oct 3 1998	mod_autoindex	Directories have size shown as “0k” instead of “-” in Fancy Heading.
#5389	Oct 29 1999	mod_rewrite	mod_rewrite is *SEVERELY* broken by a one-character bug introduced in version 1.148. The bug causes the next-to-last backref substitution to never happen... if you only have one backref, the $1 disappears without a trace!
#10090, #10185	Mar 14 2002	mod_rewrite	rnd map type balancing broken; ReWriteMap MapType ’rnd’ not working.

Table 1: Latest four problem reports in Apache that were solved with a commit that included the following keywords: *log*, *introduced* and *PR* followed by a number. The date reported, category and main description come from Apache’s GNATs defect system.

PR #3130. The PR #3130 documents a failure which affected the `mod_autoindex` module. We perform the following steps as described in our method:

1. *Find root of CGI.* For this module, its main entry point function is `handle_autoindex()`, therefore we use this function as the root of the CIG.
2. *Determine date at which the CIG is created.* We compute the CIG for the date the defect was reported, Oct. 3.
3. *Determine a pruning area.* The defect (*showing size “0k” instead “-” for directories when listing the contents of a directory*) seems to be specific to this Apache module; therefore, a good area to concentrate on is the source code of the module—the file `mod_autoindex.c`.
4. *Determine a period of interest.* The submitter of the report claimed that the defect was not present in version 1.2.6 but occurred in every one of the 1.3.x versions. Version 1.3.0 was released on June 1, 1998; this date will become the upper limit of the period of interest (the defect is known to be present at this date). The lower limit (when the bug is introduced) is more difficult to determine. Version 1.2.6 was developed in parallel to 1.3.x (1.2 was in maintenance mode while 1.3.x was being started). This meant that we could not use the date of the release of 1.2.6 as a starting date for period of interest. This defect could have been inserted early in the development of versions 1.3.x The Change Impact Overview of `handle_autoindex()` is shown in Figure 14. There is a remarkable growth in the number of changed functions on day 50 (an increase of 75 functions). This was surprising. We explored the version logs to find out why so many functions had been changed and found the reason: on April 11, 1998 there was a major renaming of functions and variables in Apache[2]. Including this day will result in a CIG that has most of its nodes marked as changed.

[2]We do not currently deal with function renames; we consider the function with the old name deleted and one added with the new name, and the function that was modified –usually only a token replace to reflect the change in name of the called function– changed; this is an area that needs further work.

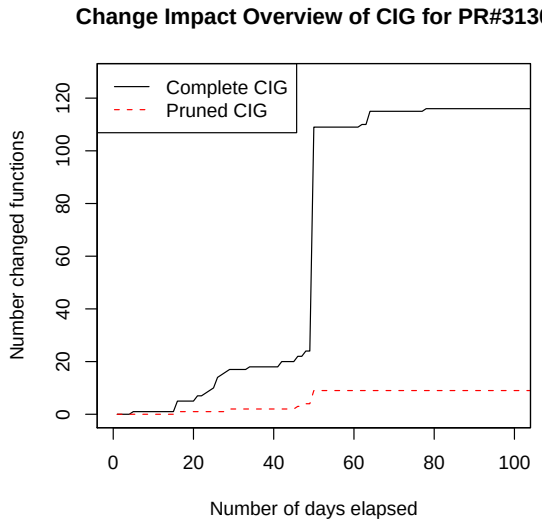

Figure 14: Change Impact Overview of `handle_autoindex()` showing the impact of changes older than June 1, 1998. As it can be observed, day 50 (April 11, 1998) had a sharp increase in changed functions (75 functions).

Figure 15 shows the CIG of `handle_autoindex()` when the changed of the last 100 days are considered; 89% of its nodes are changed. We illustrate the use of annotations with this graph in Figure 16. In this CIG, paler nodes depict older changes (this information was extracted automatically from the history of changes during the creation of the CIG). As it can be seen, most changes show a pale red (most of them changed during the rename), but many others are bright red (modified after the rename).

Hence, to avoid the effect of the rename, we recomputed the CIG with a period of interest between April 12 and June 1, 1998 (49 days). The corresponding CIG is shown in Figure 17. There is still a significant number of changed functions during this period. However, we pruned-before the graph to include only the impact of functions inside the source code of this module (file `mod_autoindex.c`) As it can be seen in Figure 14, only two functions have been changed during this period. The CIG for this period is show in Figure 18. It shows the two functions

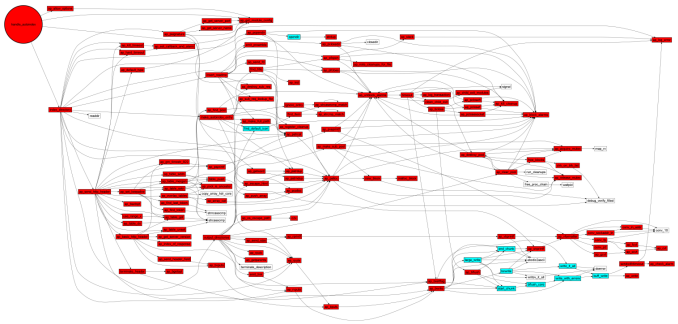

Figure 15: CIG of `handle_autoindex` (depicted as a circle) on Oct 3, 1998 showing the propagated changes for the last 100 days. Almost all nodes have been changed! Looking at the logs the answer is clear: a commit on April 11, 1998 reads "THE BIG SYMBOL RENAMING FOR APACHE 1.3". This illustrates the main limitation of CIGs: if too many functions change most of the graph is annotated.

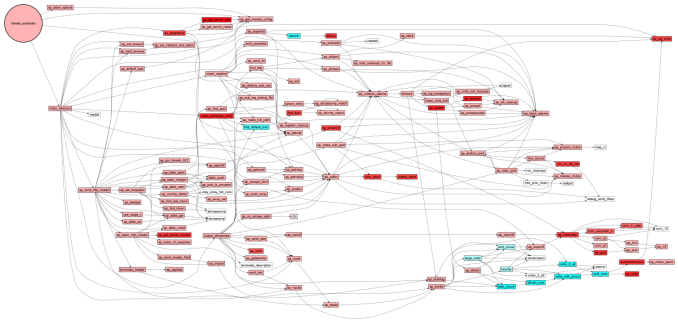

Figure 16: CIG of Figure 15. It has been annotated to show the age of the last change: paler red nodes were modified less recently.

inside `mod_autoindex.c` that were changed during the period of interest (April 12 to Jun 1, 1998). The change to one of them (`make_autoindex_entry`) introduced this defect. We created and pruned four CIGs during the investigation of this bug. The function that is the source of the defect is marked in all four CIGs.

PR #5389. As before, we start by estimating the basic parameters for the CIG.

1. *Find root of CGI.* The module `mod_rewrite` contains three functions that are entry points to the module. Of them `hook_uri2file()` appears to the only one relevant as the one where the defect appears. We will use this function as the root of the CIG.
2. *Determine date at which the CIG is created.* We compute the CIG for the date the defect was reported, Oct 29, 1999.
3. *Determine a pruning area.* This defect appears to affect only the `mod_rewrite` module. We will prune the graph to include only functions in the file that contains it: `mod_rewrite.c`.
4. *Determine a period of interest.* The defect report indicated that the bug was introduced in revision 1.149, which was performed in Oct 27, 1999. We will consider only changes until Oct 27, 1999 (inclusive). The Change Impact Overview of this CIG shows that on day 7 there were some changes that affected a large number of functions (an increase in 13 changed functions that day). On the other hand, during the first 5 days only functions within the module's source code have been changed (i.e., the number of changed functions in the pruned area is the same as in the complete graph). For the sake of illustrating CIGs we will consider changes during the period of Oct 22 to Oct 27, 1999 (four functions changed in `mod_rewrite.c` plus one changed somewhere else.

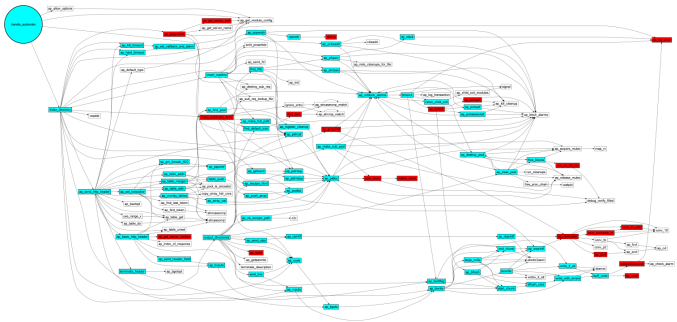

Figure 17: CIG of `handle_autoindex` on Oct 3, 1998 showing the propagated changes from April 12 to Jun 1, 1998.

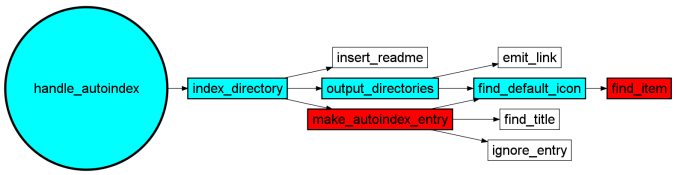

Figure 18: The pruned-before CIG of `handle_autoindex` on Oct 3, 1998 showing the propagated changes from April 12 to Jun 1, 1998. The CIG has been pruned to include only the impact of functions inside `mod_autoindex.c` (where this Apache module is implemented). The defect was found in one of the two functions marked as changed (`make_autoindex_entry`).

Figure 20 shows the CIG of `hook_uri2file()`, computed on Oct. 29, 1999, and including changes from Oct 22 to Oct 27, 1999. As expected, there are only five functions changed in the system that affect `hook_uri2file` (which had changed as well). One of the these functions, `ap_write`—the red node located at the right of the graph—propagates through a large proportion of the graph. Figure 21 shows the pruned-after CIG of `handle_autoindex` showing only functions inside its Apache module (`mod_autoindex.c`); this CIG shows the impact of the changes to `ap_write`, even when its node is no longer present. This defect was found to be inside one of the functions marked as changed: the function `expand_backref_inbuffer` had large sections rewritten on Oct 27.

PRs #10090 and #10185. These two bug reports documented a failure that affected the rewrite module. Again, we follow the steps of our method to create the CIGs:

1. *Find root of CGI.* Because the defect appears in the same module as #5389, and using the same rational, we use the function `hook_uri2file()` as the root of the CIG, .

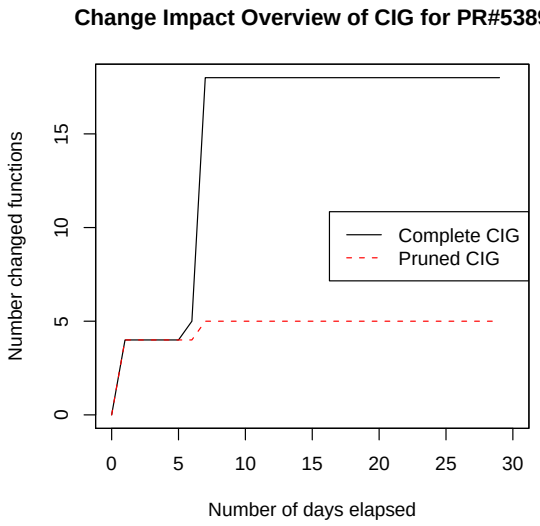

Figure 19: Change Impact Overview of `hook_uri2file()` showing the impact of changes older than Oct 27, 1999.

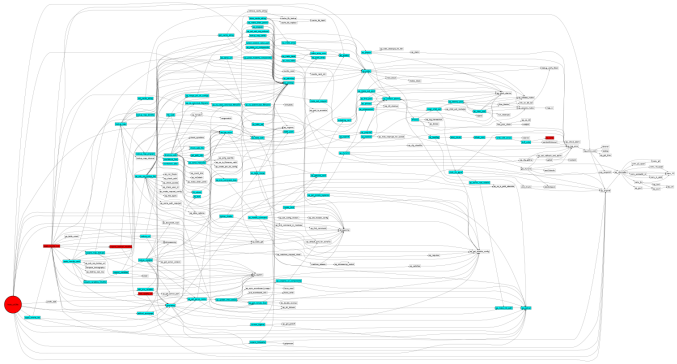

Figure 20: CIG of `hook_uri2file` on Oct 29, 1999 showing the propagated changes for the last 5 days. The changes to the function `ap_write` (rightmost red node) have propagated to a large portion of the graph.

2. *Determine date at which the CIG is created.* The defects were reported March 14, 2002. This is the date at which the CIG is created.
3. *Determine a pruning area.* Like in PR #5389, we prune functions outside the source code of the module, the file `mod_rewrite.c`.

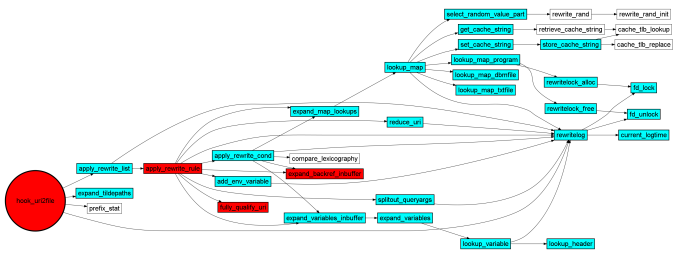

Figure 21: The pruned-after CIG of `hook_uri2file` on Oct 29, 1999 showing the propagated changes for the last 5 days, and only functions in `mod_rewrite.c`. The failure described in PR#5389 was found in `expand_backref_inbuffer` (rightmost red function).

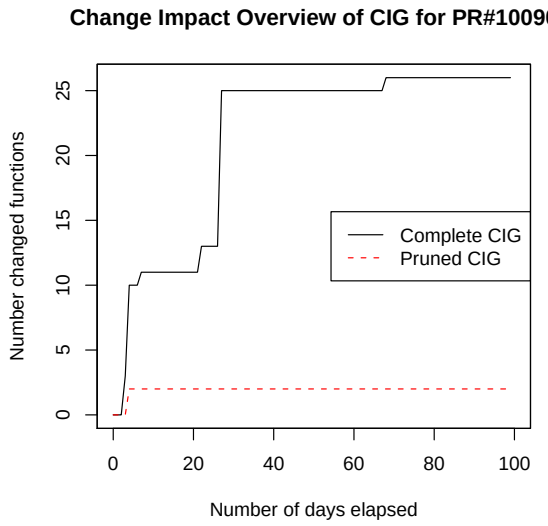

Figure 22: Change Impact Overview of `hook_uri2file()` showing the impact of changes older than Jan 24; after date 95—Oct 12, 2002—it is known that the defect does not exist. Day 4 shows the only jump in the number of changed functions in the pruned CIG, when two functions are changed.

4. The submitter of one of these reports claimed that a change between versions 1.3.22 (Oct 12, 2001) and 1.3.23 (released Jan 24, 2002) had broken the "rand map type". These two dates delimit the period of interest. But inspecting the Change Impact Overview, depicted in Figure 22, one can observe that after day 4 (corresponding to Jan 20, 2002) the number of changed functions in the CIG remains constant. This means that any pruned CIG created with a period of interest that starts any day between Oct 12 and Jan 20, and ends in Jan 24, will contain the same changed functions.

Figure 23 shows the CIG of `hook_uri2file()`, computed for the period Jan 20 to 24, 2002. Only 10 functions are modified during this period, but most of them are outside the module's source code. Figures 24 and 25 show the CIGs pruned-after and pruned-before, respectively, to exclude functions outside the module. The main difference between both is that the pruned-after CIG shows the impact of changes outside the pruning area (it shows functions affected by changes to changed functions not in the module); the pruned-before, on the other hand, only shows functions affected by changed functions inside the module. Finally, Figure 26 shows the pruned-before CIG with unaffected nodes removed. Removing unaffected nodes makes the CIG easier to read, and simplifies rendering. Only two functions have been changed, one of them `rewrite_rand`, where the defect was introduced. This change took place on Jan 20, 2002, just 4 days before the release of 1.3.23.

The impact-annotated source code of `rewrite_rand` for this period is presented in Figure 27[3]. The error was introduced

[3] We build it manually, from the diffs of the changes in question.

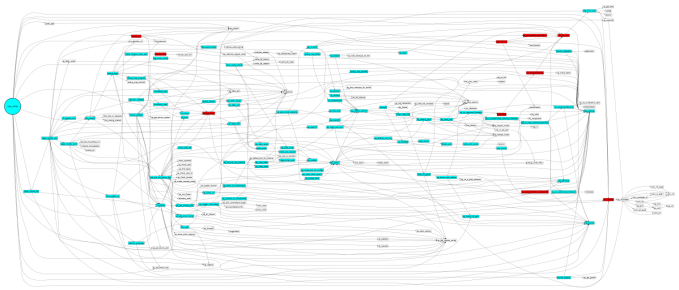

Figure 23: CIG of `hook_uri2file()` on March 14, 2002, showing the propagated changes from Jan 20 to 24, 2002.

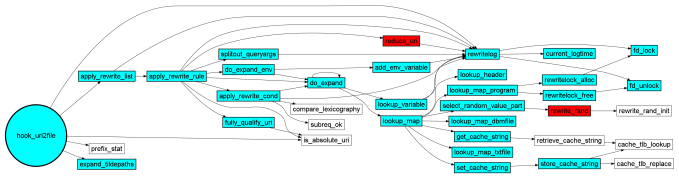

Figure 24: CIG of `hook_uri2file()` on March 14, 2002, showing the propagated changes made from Jan 20 to 24, 2002, and pruned-after to include only functions inside `mod_rewrite.c`. This CIG shows functions affected by changes outside `mod_rewrite.c`.

when a developer added the typecast (int) to the front of the expression; the priority of this operator applied the typecast to the first part of the expression only. The log of this change reads: "Dispatch 26 compiler emits into oblivion. Vetting is desired, please post to the list if you participate. They are all blindingly obvious, but extra eyes always help. This eliminates all but the regex emits and MSVC's borked misdeclaration of FD_SET.".

Changes like these are probably riskier than traditional changes because they are done in mass (26 compiler errors fixed in one change). It is clear that the developer did not fully test this change. Otherwise the bug would have been discovered almost immediately; instead, the failure was reported almost three months after the bug was introduced.

Impact-annotations can be very useful in these situations. Right after the change is committed, certain developers can be informed that the code they are responsible for could be affected by such commit, and they might be more inclined to check it for correctness. Otherwise, as in the case of this bug, nobody reviewed this line of code (or if it was reviewed, the reviewer failed to catch the bug).

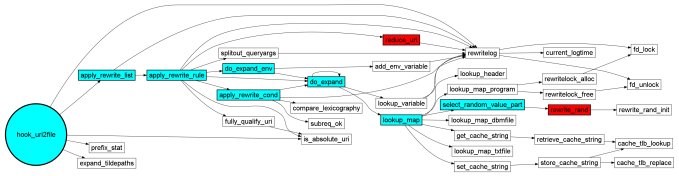

Figure 25: CIG of `hook_uri2file()` on March 14, 2002, showing the propagated changes made from Jan 20 to Jan 24, 2002, and pruned-before to include only functions inside `mod_rewrite.c`.

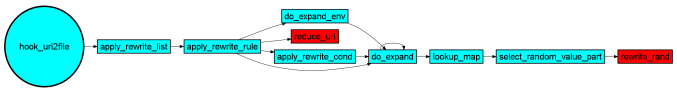

Figure 26: Same CIG as Figure 25 after unaffected functions have been removed. This often simplifies the rendering of the CIG making it more readable. The failure described in PRs #10090 and #10185 was found in `rewrite_rand()`, the rightmost node.

5. Related Research

Change propagation is a central activity during software development. As developers modify code to introduce new features or fix bugs, they must ensure that other parts of the software system are updated to be consistent with these new changes. For example, if the interface for a function changes, its callers have to be modified to reflect the new interface, otherwise the source code won't compile nor link.

Many hard to find bugs are introduced by developers who did not notice dependencies between entities, and failed to propagate changes correctly. Our proposal provides a practical and simple method that mines historical code changes to help maintainers in fixing bugs caused by mis-propagation of changes.

Many researchers note the dangers of mis-propagating changes. For example, Parnas tackled the issue of software aging and warned of the ill-effects of *Ignorant Surgery*, code changes done by developers with limited knowledge of the system [14]. Arnold and Bohner give an overview of several formal models of change propagation [15, 16]. The models propose several tools and techniques that are based on code dependencies and algorithms such as slicing and transitive closure [2, 1] to assist in code propagation. Rajlich proposes another formal model for change propagation [17], given a particular change request Rajlich's model can be used to guide developers in propagating the change in a systematic manner. These models help developers avoid mis-propagating changes. In contrast, our model helps developers identify possible mis-propagation of changes when fixing bugs.

Several researchers have proposed the use of historical data related to a software system to assist maintainers of large software systems. Cubranic *et al.* present a tool which uses bug reports, news articles, and mailing list messages to suggest pertinent software development artifacts [18]. Chen *et al.* attach the comments associated with source code changes to each code statement and use these comments to index the code and help in locating the lines of code associated with a particular feature [19]. Hassan and Holt propose annotating the dependency graph of a software system with historical information to assist in understanding the rationale for the current design [20]. Mockus *et al.* use historical code changes to help identify code experts based on prior changes for a particular code segment [21]. Relative to previous work on the use of historical information we recognize the importance of historical information and we integrate the historical information into the commonly used dependency information (i.e., the dependence graph).

Much of the intuition and driving force behind our work stems from the following two related works. Graves *et al.* show that surprisingly most bugs are not due to complex code

```
static int rewrite_rand(int l, int h) {
  rewrite_rand_init();
  /* Get [0,1) and then scale to the appropriate range.  Note that using
  * a floating point value ensures that we use all bits of the rand()
  * result.  Doing an integer modulus would only use the lower-order bits
  * which may not be as uniformly random.  */
  return (int)((double)(rand() % RAND_MAX) / RAND_MAX) * (h - l + 1) + l;
}
```

Figure 27: Annotated source code of `rewrite_rand_init`. Its first source code line was not modified nor affected; the second—the cause of the failure– was modified on Jan 20, 2002, when the typecast operator `(int)` was inserted, truncating the first part of the expression instead of the entire result. The log of the change explains: "Dispatch 26 compiler emits into oblivion. Vetting is desired... They are all blindingly obvious, but extra eyes always help...".

instead they are usually due to frequently changing code [22]. Given the location of a reported bug, our method flags statements which depend directly or indirectly on changing code. Sliwerski *et al.* present a procedure which identifies risky code regions using information from version history and from the bug tracking system [23]. They present an `Eclipse` plug-in which informs developers about the risk of a location on a statement basis. The risk is calculated based on the number of times a particular statement was part of a change that was later identified as being a buggy change. Similar to Sliwerski *et al.*, developers could use our method to identify risky parts of the code. In contrast, our definition of risk is a second-order definition: instead of identifying risky code, we identify code that depends on risky code by, for example, calling code which tends to have many buggy changes.

Delta debugging is an algorithm proposed by Zeller and Cleve to identify the piece of code executed that caused a failure [24, 25]. Therefore relevant variables and values involved in the error are isolated and state differences of a run where the failure occurs and of a run where the failure does not occur are obtained. The moment when the piece of code that causes the failure is executed points to the bug which must be fixed. Delta requires the existence of a test suite, which are not frequently available. In contrast, our method uses historical code change information to direct the attention of developers to the most likely change that might have caused the bug.

6. Discussion

6.1. Limitations

There are three major shortcomings of our method:

- A single commit can result in too many marked nodes in the dependence graph, becoming impractical—as shown in Figure 15 where 81.7% of the functions have been changed).
- It is sometimes not easy to determine the period of interest for which the dependence graph should be created. The developer needs to experiment and apply her experience and insight in the selection of the period of interest. The Change Impact Overview visualization (as shown in Figure 14), along with the logs of the changes that affect those functions can be used to narrow the period. Also, the larger the period of interest, the more likely that changes that are not relevant are included in the CIG.
- Given the recursive nature of CIGs, the larger the CIG (in terms of nodes) the higher the probability that it includes a function is marked as changed that has nothing to do with the defect in question.

The larger the graph and the number of changed functions in it, the more difficult it will be for the developer to find the source of the defect.

Systems with a very good suite of tests will benefit from CIGs. Failures are likely to be found early, making the period of observation very small. The automatic annotations will point to the few areas of the system that are likely to have changed in such a small period.

Another method to deal with changes that affect many functions is to select only a subset of changes based on certain criteria–as described in [26]. For example, "select all commits during the period of observation except the one that renamed all symbols". The risk of using this method is that one might inadvertently skip the commit that introduced the bug which caused the reported failure. This is not an issue when one is interested only in being aware of what areas of the system have changed (and which have been affected). For example, a developer might be interested to get an idea of what areas have been affected by the changes performed by another developer; in this case the criteria is to select only the changes authored by the latter author.

Developers won't be able to use our proposed method to resolve every reported fault. Instead as we observe in the case study, our method is most suitable for bugs that appear due to earlier changes in a software application. *This-Worked-Before* bugs are probably the best way to describe the bugs that would benefit the most of our method. Bugs which are due to unexpected usages of an application, or changes to its environment won't benefit from our method. Our method is one of the many tools available for developers who are working on large code bases, and extends and enhances commonly used basic dependence graphs by incorporating historical information to prune and highlight the graphs.

6.2. Extraction of the Dependence Graphs

The effectiveness of CIGs depends heavily on the quality of the extraction of the dependence graphs from the source code

of the system. In our current implementation we use a simple fact extractor that does not take into account function pointers nor polymorphic function calls. Our method to create CIGs, however, can work with any dependence graph extractor that generates a graph where functions are represented as nodes, and function calls as edges.

Extending our extraction and method to object-oriented languages is yet to be done. Many of the general ideas should follow to object-oriented systems. However, we still need to explore the limitations and benefits on a real large scale object-oriented system.

6.3. Effectiveness of CIGs

We view CIG as a tool in a larger toolset that developers can use to locate bugs in an efficient manner. Other tools in the toolset could be code slicers, debuggers, and basic dependency browsers.

The number of bugs that would benefit from our approach is highly dependent on the experience of the developer using the CIG, the application at hand, and the reason for the occurrence of particular bugs (i.e., did they occur due to prior changes or due to changes in the usage patterns of the application?). CIGs are mainly used to fix bugs due to prior changes. Determining such bugs is an open research problem that continues to be investigated by others, such as Kim et al. [27].

A user study is needed to study the true effectiveness of CIGs. For this paper, we chose to perform an exploratory case study by picking several real-life bugs and showing how non-experts, like us, could fix these bugs with limited knowledge of Apache. A user study would require us to either recruit real-life developers working on such a large system, or to study a much smaller system. Recruiting such developers is usually very hard and not feasible; and a smaller case study conducted by students would limit the scope of our findings.

The examples above are too few and lack the necessary rigor to be considered a formal evaluation of CIGs. At its most basic form, a CIG is just a size-reduced dependence graph that can be very rapidly calculated. Therefore developers can explore the use of CIGs without requiring any additional commitment on their behalf. The CIG method could be integrated as part of commonly available dependence browsers, e.g., the dependency browser in a IDE.

Our case study examples demonstrate the ability of the a CIG to narrow the search space for the source of a failure by highlighting areas of the code that have changed and that might have an impact on a failing function. We expect that focusing the attention of developers to specific areas of the code will reduce the time needed to investigate a bug.

Table 2 shows the ratios of changed and affected functions for each of the CIGs presented in our case study. Although the CIGs are relatively large, the ratio of changed nodes is small (as small as 2.3%) for most of the graphs, and some of them have very few nodes. Ratios of affected functions shown table 2 may in some cases be larger than the ratio of changed functions because of pruning.

However even if a graph contains few changed nodes, the number of affected nodes (functions where a failure can occur) can be large. In other words, a bug introduced in a function has the potential to present itself as a failure in many other functions.

PR	CIG in Figure	Total Nodes	Ratio Affected	Ratio Changed	LOCs
#1352	12	203	30.0%	18.2%	1647
	13	4	0%	50%	172
#3130	15	142	6.8%	81.7%	3055
	17	142	45.8%	16.9%	1287
	18	10	40.0%	20.0%	360
#5389	20	206	43.2%	2.4%	545
	21	36	69.4%	11.1%	526
#10090,	23	191	44.5%	5.3%	339
10185	26	10	80 %	20.0%	61

Table 2: Effectiveness of the CIGs for the examples presented in our case study. The last column, LOCs, is the sum of LOCs of the functions that were changed.

6.4. Improving the tracking of a function's evolution

Some functions are renamed, merged, split or their code cloned. We believe it will be worthwhile to track this evolution and use the resulting information in the creation of CIGs.

Similarly, the analysis we present relies on a textual comparison with comments removed, code re-indented and code renamed. A more powerful approach would involve comparing the Abstract Syntax Trees (ASTs) of the function before and after the change (using methods such as [28]). Did the change affect the AST of the code? Was it a change to a constant (such as a string to be printed)? Was it a change to a token (perhaps the result of a rename of a function in the same commit). This information could be used to include and exclude some changes when building a CIG. However an AST-based approach might limit our ability to study all historical changes since there some historical snapshots might contain code that is not compilable.

6.5. CIGs and slicing

A CIG reduces the size of a dependence graph using various pruning techniques. Dependence graph represent the state of practice in investigating code changes by developers. On the other hand, code slicing techniques represent the state of the art with slices being more precise and supporting various pruning techniques based on data and control flow characteristics.

Code slicing provides a more in-depth analysis of impact of the changed code. Slices track indirect calls via parameters, and changes to variables, in contrast to our method which is based only on tracking function calls. However, the benefits of a slice come at a high cost—with slicing techniques requiring extensive calculation time. For instance, Binkley et al. [5] show that a slice in a 150 kLOC program can take up to 118 hours to calculate. In contrast the CIGs are not as precise but can be built very quickly requiring milliseconds to calculate. The fast calculation speed ensures that we can integrate CIGs as part of the daily toolset used by developers. Developers could explore various CIGs with little time commitment. In future work, we

would like to explore a more detailed comparison of CIGs versus slicing. We also want to explore combining CIGs and slicing to build on the strengths of both approaches: the accuracy of a slice, and the speed of a CIG.

6.6. Improving the automatic annotations

The *changed* functions in a dependence graph can be further annotated with a measure of the change, such as the number of LOCs changed, the difference of the complexity between before and after, a likelihood that the change is a risky one (based on the type of change, who made the change, when the change was performed, etc.). Such information can then be propagated to the callers.

6.7. Support for automatic annotations during editing/debugging of source code

The annotated source code could be computed on-demand within a typical IDE (such as `Eclipse`) or a debugger. In a preparation stage, the history of the project is analyzed, and the change history of each function is created. The latest dependence graph of the system is computed. At this point it is possible to incrementally continue updating the change histories of functions and the latest dependence graph as the version control system detects a new source control change. When perusing source code, the developer will select the period of interest (either by time, or by specifying two different changes). If the code being browsed has not changed (with respect to the latest version in the source control repository), the pre-computed CIGs would be used, otherwise a new one will be computed. The source code will be annotated automatically using these CIGs. From our experience we know that a CIG can be built within milliseconds for a very large project – making a CIG an interactive and responsive tool that developers can use on a daily basis.

7. Conclusions

All too often developers must investigate failures in functions and features that have not changed. Investigating such failures is challenging and time consuming since these failures are occasionally due to bugs introduced by prior code changes. In this paper we present a method which guides developers in their investigation of such failures by annotating the dependence graph and the source code of a function with the impact of prior historical changes. Using the annotation, developers can quickly pinpoint the changes which most likely introduced the bug, causing the reported failure. We demonstrate the feasibility of our method through an exploratory case study on the Apache Web Server. Our method permits developers to considerably prune the size of the investigated dependence graph. With a smaller number of nodes to investigate, developers can better focus their attention to the nodes that are most likely the cause for the failure.

Acknowledgements

We would like to thank our anonymous reviewers for their helpful comments on an early version of this paper. The work of D. M. German and A. Hassan is funded in part by the Natural Sciences and Engineering Research Council of Canada. The work of G. Robles has been funded in part by the European Commission, under the FLOSSMETRICS (FP6-IST-5-033547), QUALOSS (FP6-IST-5-033547) and QUALIPSO (FP6-IST-034763) projects, and by the Spanish CICyT, project SobreSalto (TIN2007-66172).

References

[1] M. Weiser, Programmers use slices when debugging, Commun. ACM 25 (7) (1982) 446–452.

[2] M. Weiser, Program slicing, in: Proceedings of the International Conference on Software Engineering (ICSE 1981), 1981, pp. 439–449.

[3] D. Binkley, M. Harman, A large-scale empirical study of forward and backward static slice size and context sensitivity, in: ICSM '03: Proceedings of the International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, 2003, p. 44.

[4] D. Binkley, N. Gold, M. Harman, An empirical study of static program slice size, ACM Trans. Softw. Eng. Methodol. 16 (2) (2007) 8. doi:http://doi.acm.org/10.1145/1217295.1217297.

[5] D. Binkley, M. Harman, J. Krinke, Empirical study of optimization techniques for massive slicing, ACM Trans. Program. Lang. Syst. 30 (1) (2007) 3. doi:http://doi.acm.org/10.1145/1290520.1290523.

[6] H. Agrawal, J. R. Horgan, Dynamic program slicing, in: PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation, ACM, New York, NY, USA, 1990, pp. 246–256. doi:http://doi.acm.org/10.1145/93542.93576.

[7] X. Zhang, N. Gupta, R. Gupta, A study of effectiveness of dynamic slicing in locating real faults, Empirical Software Engineering 12 (2) (2007) 143–160. doi:http://dx.doi.org/10.1007/s10664-006-9007-3.

[8] T. Kamiya, S. Kusumoto, K. Inoue, Ccfinder: a multilinguistic token-based code clone detection system for large scale source code, IEEE Trans. Softw. Eng. 28 (7) (2002) 654–670. doi:http://dx.doi.org/10.1109/TSE.2002.1019480.

[9] D. M. German, A study of the contributors of PostgreSQL, in: 3rd International Workshop on Mining Software Repositories–MSR Challenge Reports (MSR 2006), 2006.

[10] P. Weißgerber, S. Diehl, Identifying refactorings from source-code changes, in: 21st IEEE/ACM International Conference on Automated Software Engineering, 2006, pp. 231–240.

[11] M. Kim, D. Notkin, D. Grossman, Automatic inference of structural changes for matching across program versions, in: Proceedings of the 29th International Conference on Software Engineering (ICSE 2007), IEEE Computer Society, 2007, pp. 333–343.

[12] L. Zou, Using origin analysis to detect merging and splitting of source code entities, IEEE Trans. Softw. Eng. 31 (2) (2005) 166–181, member-Godfrey,, Michael W. doi:http://dx.doi.org/10.1109/TSE.2005.28.

[13] A. Mockus, R. T. Fielding, J. D. Herbsleb, Two case studies of Open Source software development: Apache and Mozilla, ACM Transactions on Software Engineering and Methodology 11 (3) (2002) 309–346.

[14] D. L. Parnas, Software aging, in: Proceedings of the International Conference on Software Engineering (ICSE 1994), Sorrento, Italy, 1994, pp. 279–287.

[15] R. Arnold, S. Bohner, Impact analysis - toward a framework for comparison, in: IEEE International Conference Software Maintenance (ICSM 1997), Montréal, Quebec, Canada, 1993, pp. 292–301.

[16] S. Bohner, R. Arnold, Software Change Impact Analysis, IEEE Computer Soc. Press, 1996.

[17] V. Rajlich, A model for change propagation based on graph rewriting, in: IEEE International Conference Software Maintenance (ICSM 1997), Bari, Italy, 1997, pp. 84–91.
URL `citeseer.nj.nec.com/rajlich97model.html`

[18] D. Cubranic, G. C. Murphy, Hipikat: Recommending pertinent software development artifacts, in: Proceedings of the 25th International Conference on Software Engineering (ICSE 2000), ACM Press, Portland, Oregon, 2003, pp. 408–419.

[19] A. Chen, E. Chou, J. Wong, A. Y. Yao, Q. Zhang, S. Zhang, A. Michail, CVSSearch: Searching through source code using CVS comments, in: IEEE International Conference Software Maintenance (ICSM 2001), Florence, Italy, 2001, pp. 364–374.
URL citeseer.nj.nec.com/436456.html

[20] A. E. Hassan, R. C. Holt, Using development history sticky notes to understand software architecture, in: IWPC, 2004, pp. 183–193.

[21] A. Mockus, L. G. Votta, Identifying reasons for software changes using historic databases, in: Proc Intl Conf Softw Maintenance, 2000, pp. 120–130.

[22] T. L. Graves, A. F. Karr, J. S. Marron, H. P. Siy, Predicting fault incidence using software change history, IEEE Trans. Software Eng. 26 (7) (2000) 653–661.

[23] J. Sliwerski, T. Zimmermann, A. Zeller, Hatari: raising risk awareness, in: ESEC/SIGSOFT FSE, 2005, pp. 107–110.

[24] A. Zeller, Isolating cause-effect chains from computer programs, in: SIGSOFT FSE, 2002, pp. 1–10.

[25] H. Cleve, A. Zeller, Locating causes of program failures, in: 27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA.

[26] A. McNair, D. M. German, J. Weber-Jahnke, Visualizing software architecture evolution using change-sets, in: "Proc. 14th Working Conference on Reverse Engineering", 2007, pp. 140–149.

[27] S. Kim, T. Zimmermann, K. Pan, E. J. J. Whitehead, Automatic identification of bug-introducing changes, in: ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, Washington, DC, USA, 2006, pp. 81–90. doi:http://dx.doi.org/10.1109/ASE.2006.23.

[28] B. Fluri, M. Wuersch, M. PInzger, H. Gall, Change distilling: Tree differencing for fine-grained source code change extraction, IEEE Trans. Softw. Eng. 33 (11) (2007) 725–743. doi:http://dx.doi.org/10.1109/TSE.2007.70731.