HARVARD | BUSINESS | SCHOOL

Exploring the Relationship Between Architecture Coupling and Software Vulnerabilities

Robert Lagerström
Alan MacCormack
Lee Doolan
Carliss Baldwin
Dan Sturtevant

Working Paper 18-031

Exploring the Relationship Between Architecture Coupling and Software Vulnerabilities

Robert Lagerström
KTH Royal Institute of Technology

Carliss Baldwin
Harvard Business School

Alan MacCormack
Harvard Business School

Dan Sturtevant
Silverthread Inc.

Lee Doolan
Silverthread Inc.

Working Paper 18-031

Exploring the Relationship Between Architecture Coupling and Software Vulnerabilities

Robert Lagerström[1,2(✉)], Carliss Baldwin[2], Alan MacCormack[2], Dan Sturtevant[3], and Lee Doolan[3]

[1] KTH Royal Institute of Technology, Stockholm, Sweden
robertl@kth.se
[2] Harvard Business School, Boston, USA
[3] Silverthread Inc., Boston, USA

Abstract. Employing software metrics, such as size and complexity, for predicting defects has been given a lot of attention over the years and proven very useful. However, the few studies looking at software architecture and vulnerabilities are limited in scope and findings. We explore the relationship between software vulnerabilities and component metrics (like code churn and cyclomatic complexity), as well as architecture coupling metrics (direct, indirect, and cyclic coupling). Our case is based on the Google Chromium project, an open source project that has not been studied for this topic yet. Our findings show a strong relationship between vulnerabilities and both component level metrics and architecture coupling metrics. 68% of the files associated with a vulnerability are cyclically coupled, compared to 43% of the non-vulnerable files. Our best regression model is a combination of low commenting, high code churn, high direct fan-out within the main cyclic group, and high direct fan-in outside of the main cyclic group.

Keywords: Security vulnerabilities · Software architecture · Metrics

1 Introduction

Cyber security incidents and software vulnerabilities cause big problems with increasing societal impact. Both individual home users and large corporations face similar problems with exploited software vulnerabilities leading to loss in confidentiality, integrity, and availability, and at the end of the day - time and money. Many seem to agree that software architecture complexity is a key issue when it comes to software vulnerabilities. Quoting a well-cited blog post by Bruce Schneier[1] "The worst enemy of security is complexity". The basic argument is that poorly designed and maintained software systems tend to embed highly complex code and architectures, which in turn increase the likely occurrence of vulnerabilities waiting to be exploited. However, few studies have explored the relationship of complexity to vulnerabilities and the findings to this point are inconclusive [1, 2] and far from generalizable.

[1] www.schneier.com/essays/archives/1999/11/a_plea_for_simplicit.html.

E. Bodden et al. (Eds.): ESSoS 2017, LNCS 10379, pp. 53–69, 2017.
DOI: 10.1007/978-3-319-62105-0_4

Some existing studies of software vulnerabilities include direct coupling as a predictive variable, e.g. [3–5]. However, to our knowledge, other coupling measures such as indirect coupling and cyclic coupling have not been tested in relation to vulnerabilities. These measures have been shown to affect other software performance outcomes such as defects e.g. [6], productivity e.g. [7], and maintenance cost e.g. [8]. Our theory is that this is also the case for software vulnerabilities.

In this paper, we measure (and visualize) the Google Chrome software architecture and explore the correlation between software vulnerabilities and component-level metrics and different architecture coupling measures. Our component metrics include: code churn, source lines of code, cyclomatic complexity, and comment ratio. Our coupling measures fall into three categories: direct coupling (fan-in & fan-out), indirect coupling (fan-in & fan-out), and cyclic coupling. Studying 16,268 C-files of the March 2016 Chrome release and linking them to 290 files that were changed in order to fix 185 vulnerabilities, we found that both component metrics and the different coupling measures are significantly correlated with vulnerabilities. However, due to limitations of the sample, when testing a set of regression models, we could not untangle the impact of the different coupling measures.

Our main contribution in this paper is to add new findings to the work on software metrics and vulnerabilities, bringing the field closer to generalizable and conclusive results. To this end, we focus on the Chromium project, which has not been studied from the perspective of vulnerabilities. Our correlations are both strong and significant. They indicate that architectural coupling measures might be used in addition to component-level metrics to improve vulnerability prediction models, although this needs additional exploration.

The rest of this paper is organized as follows: Sect. 2 presents related work in three areas; software metrics and defects, software metrics and vulnerabilities, and impact of architectural coupling on different performance measures. In Sect. 3 we describe our measures of complexity and coupling. The Chromium project is described in Sect. 4, followed by our analysis of Chrome and software vulnerabilities in Sect. 5. Our study and potential future work are discussed in Sect. 6. Section 7 concludes the paper.

2 Related Work

For purposes of exposition, we have divided related work in three categories: studies relating software metrics and defects; studies relating software metrics and vulnerabilities; and studies relating architecture coupling measures to different performance outcomes.

2.1 Software Metrics and Defects

Numerous studies have looked at the relationship between software metrics and defects. We discuss those that use measures most closely related to the ones we test. For the interested reader [9, 10] present comprehensive literature studies on this topic.

In [11] Kitchenham et al. found that the number of files used by a given file, a coupling measure we label "direct fan out," is associated with defects, although source lines of code, a code measure, was a stronger indicator. A similar study by Basili et al. [12] found the opposite: that coupling measures were better able to predict faults traditional code metrics, such as lines of code. In [13], Nagappan and Ball showed that code churn, a relative measure of change in a file, is an early indicator of defect density. Schröter et al. looked at usage dependencies, a form of coupling between components and showed that these are good predictors of defects [14]. Zimmermann and Nagappan studied network measures of coupling, such as density and centrality, for defect prediction and found that these perform better than other complexity metrics [15]. Steff and Russo then showed that dependency changes are strong defect indicators [16].

From these and other studies, it appears that both coupling measures and component-level metrics have proven successful in defect prediction. Since vulnerabilities are a special class of defects, our working hypothesis is that the coupling measures and code metrics should also predict vulnerabilities.

2.2 Software Metrics and Vulnerabilities

While much attention has been paid to defects, less work has been done to examine the relationship between coupling measures and code metrics and vulnerabilities. However, elevated concerns about cyber security have brought more attention to this topic.

Neuhaus et al. studied the Mozilla project and found correlation between vulnerabilities and include statements [17]. In [18], the same authors used Red Hat to investigate the correlation between package dependencies and vulnerabilities. Zimmermann et al. also found weak correlation between a set of software metrics and vulnerabilities [19]. Nguyen and Tran used dependency graphs to look at vulnerabilities in the Mozilla Firefox Javascript Engine [20]. Shin et al. investigated the same codebase, but focused on complexity metrics [3]. Moshtari et al. [5] replicated and extended this work by including a more complete set of vulnerabilities and looking at more software applications, including Eclipse, Apache Tomcat, Firefox, Linux Kernel, and OpenSCADA. They concluded that their software (complexity) metrics are good predictors of vulnerabilities. Chowdhury and Zulkernine [21] investigated the relation between complexity, coupling, cohesion, and vulnerabilities in Mozilla Firefox. They were able to predict a majority of the files associated with vulnerabilities with tolerable false positive rates. Hovsepyan et al. [22] looked at design churn as a predictor of vulnerabilities in ten Android applications and found a statistically significant relationship between design churn and vulnerabilities in some but not all applications.

Morrison et al. [2] did not find any significant relation between complexity and vulnerabilities in their study of Microsoft products. They suggested that a set of security-specific metrics might be needed in vulnerability prediction models. Shin and Williams 1 similarly found the relationship between software complexity and vulnerabilities to be weak, and also recommended that new complexity metrics be developed for understanding security related defects.

The studies on software vulnerabilities and various metrics provide a mixed picture of the relationship. Most find some correlation or predictability, but some don't and the

overall findings are weak. Most agree that there is a need to continue exploring this topic. We note that studies to date have not looked at architectural measures in conjunction with code-level complexity metrics. In most cases, coupling is omitted as a predictive variable: when included it is limited to direct coupling.

2.3 Coupling Metrics for Outcome Prediction

In [23] MacCormack et al. used indirect coupling and cyclic coupling to measure modularity and show that modular organizations produce modular software products, verifying the so-called mirroring hypothesis. Sturtevant [7] and Akaikine [8] also studied indirect and cyclic coupling in two separate cases. They found significant differences in defect density, defect resolution time, and developer productivity as a function of coupling measures.

Baldwin et al. present empirical work using 1,286 software releases from 17 different software applications showing that most of the software systems contain one large cyclic group of interdependent files (high cyclic coupling/high levels of indirect coupling), calling it the Core [24]. Heiser et al. [25] studied cyclic coupling and indirect coupling (using the suggested method in [24]) for organizational transformation planning in a development organization. In [26], they used the same methodology to develop a strategy to prioritize software feature production.

MacCormack and Sturtevant looked at the impact of coupling (indirect and cyclic) on software defect related activity [6]. Lagerström et al. used cyclic and indirect coupling in a biopharmaceutical case to visualize and measure modularity in an enterprise architecture [27]. In subsequent work, the same authors showed that it was more costly to change software applications with many cyclic dependencies than those with few or no cyclic dependencies [28, 29].

In summary, software metrics have been successful in studies predicting the location of general defects. Work on vulnerabilities, however, is not as extensive or as conclusive, although there have been promising findings. Moreover, coupling measures have not been widely used in vulnerability studies, although they known to be correlated with other performance measures including defects. In this paper, we aim to explore software metrics including architecture coupling measures in relation to software vulnerabilities by studying the Google Chrome codebase.

3 Measuring Component and Architecture Coupling Metrics

As noted in the previous section software metrics have been used as predictors of behavior in many studies. We focus on the most common and widely used component metrics, as well as a set of architectural coupling measures.

3.1 Software Component Metrics

The most common and also the simplest software metrics is measuring source lines of code (*SLOC*), basically counting the number of lines in a software file not including

comments. This is a measure of size and has been shown to predict defects, cost and complexity e.g. [30, 31].

In 1976 McCabe proposed the cyclomatic complexity (*MCCABE*) measure [32]. It is now one of the most common complexity metrics used in software studies [9]. Basically, the McCabe metric counts the number alternative execution paths that can be followed as a program executes. Alternative paths through a procedure result from conditional branching statements (if statement, switch/case statement, while loops, et cetera). McCabe scores may be calculated for procedures (called functions in C or C++) or class methods. We calculate cyclomatic complexity for all functions and methods within a C-file, and then use the maximum figure observed within that file. In prior work, McCabe scores have been predictive of higher defect rates and lower productivity.

The comment ratio (*COMMR*) is a measure of how well commented the source code is. It is a comments-to-code ratio rather than a pure count of number of comment lines. This measure is also frequently used when analyzing software, e.g. [33]. However, there is no theoretical prediction as to correlation of comments to complexity or defects (i.e. complex or defective code may generate many comments or few comments).

Code churn (*CHURN*) measures the activity within each file in terms of number of lines of code being added, modified, or deleted. This metric is also frequently used in software studies, especially for defect prediction e.g. [13]. Recently it has proved predictive in some vulnerability studies e.g. [3].

3.2 Architecture Coupling Measures

Files in a software can be coupled in different ways: directly, indirectly, or cyclically.

Figure 1(1) represents the base case, in which the files are not coupled to any other files. In Fig. 1(2), file A is directly coupled with files B and C, i.e. B and C depend on A, but A does not depend on B and C. Thus A has a direct fan-in (*DFI*) of two and a direct fan-out (*DFO*) of zero. Modular systems theory predicts that files with higher levels of direct coupling are more defect prone, given the difficulty assessing the potential impact of changing the coupled files on the dependent files [34]. Hence we predict that coupled files would be more likely to contain vulnerabilities than a similar file with no coupling (e.g., as indicated in Fig. 1(1)). Support for such a relationship is found in empirical studies of software, in which the components are source files or classes, and dependencies denote use relationships between them [3].

Figure 1(3) depicts a more complex set of relationships between software files. File A is directly coupled to B and is indirectly coupled to files C and D (through B). That is, A has an indirect fan-in (*IndFI*) of three and an indirect fan-out (*IndFO*) of zero. In this architecture, changes may propagate between files that are not directly connected, via a "chain" of dependencies. While indirect coupling relationships are likely to be weaker than direct coupling relationships, the former are not as visible to a software developer, hence may be overlooked and more likely to produce unintended system behaviors. Measures of indirect coupling have been shown to predict both the

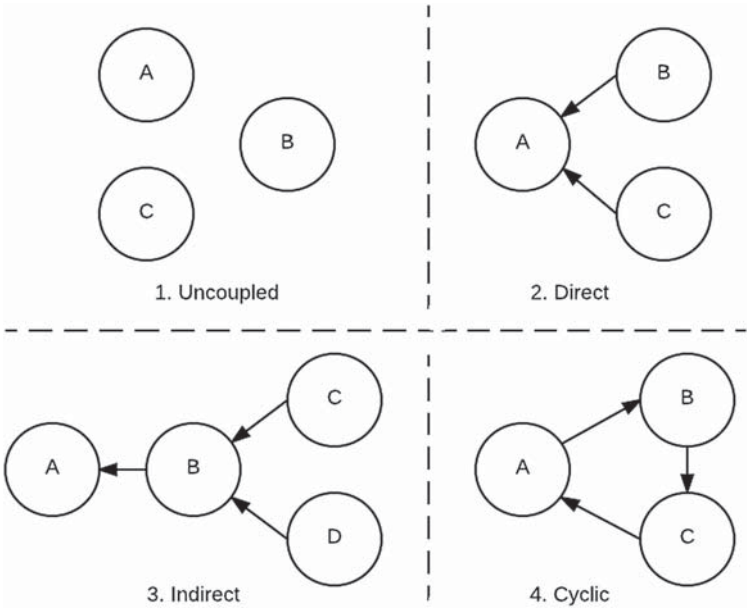

Fig. 1. Different coupling relationships between software files.

number of defects and the ease (or difficulty) with which a software system can be adapted 635.

Figure 1(4) illustrates a third pattern of coupling between files, called cyclic coupling (*CYCLIC*) 3637. In this architecture, A is coupled with C, C is coupled with B, and B is coupled with A. These files form a cyclic group – a group of files that are mutually interdependent. In contrast to Fig. 1(3), there is no ordering of these files, such that one can be changed (or developed) before the others. Rather, files in cyclic groups must often be changed concurrently, to ensure that they continue to work together effectively. When cyclic groups are large, this presents a significant challenge, increasing the likelihood of defects [35], and possibly vulnerabilities. We measure cyclic coupling as whether a file belongs to the largest cyclic group in the architecture or not, as explained in [24].

The patterns described above represent related, but conceptually distinct, patterns of coupling that exist between files. We note however, that measures of these different types are likely to be correlated. Specifically, files with high levels of direct coupling are, all else being equal, more likely to have high levels of indirect coupling. And files with high levels of indirect coupling are, all else being equal, more likely to be members of cyclic groups. It will be important to be sensitive to these issues in our empirical tests.

Table 1 presents the different types of metrics we use. It differentiates between component level metrics (source lines of code, cyclomatic complexity, commenting ratio, and code churn) and the architectural coupling measures, (direct coupling, indirect coupling, and cyclic coupling).

Table 1. Component-level and architecture coupling metrics used in this study.

Component-level metrics	Architecture coupling metrics
Source lines of code (*SLOC*)	Direct coupling (*DFI & DFO*)
Cyclomatic complexity (*MCCABE*)	Indirect coupling (*IndFI & IndFO*)
Commenting ratio (*COMMR*)	Cyclic coupling (*CYCLIC*)
Code churn (*CHURN*)	

4 The Chromium Project

The Google Chromium project[2] is an open source web-browser project from which the Chrome browser gets its source code. It is mainly written in C++ and is available for multiple platforms such as Linux, OS X 10.9 and later, Windows 7 and later, and iOS. The earliest version was released late 2008, while the first stable non-beta version (5.0.306.0) was released in early 2010. Our main focus in this study is the March 31st 2016 version called 50.0.2661.57.

4.1 Chrome Metrics and Coupling

We have measured the Chrome software architecture in terms of traditional software metrics; source lines of code (*SLOC*), cyclomatic complexity (*MCCABE*), commenting ratio (*COMMR*), and the amount of activity (*CHURN*) spent in each file to fix "regular" defects (not vulnerabilities). We also calculated the different coupling measures; direct (*DFI & DFO*), indirect (*IndFI & IndFO*), and cyclic (*CYCLIC*). All metrics and the architecture visualization (Fig. 2) were derived using a commercial analysis tool from Silverthread[3] and each metric is explained in Sect. 3. All variables are measured as positive integers, except *COMMR* which is a positive rational number and *CYCLIC* which is a binary (1/0) number.

The binary cyclic coupling metric indicates whether a file belongs to the largest cluster of cyclically dependent files (1) or not (0). The largest cyclic group is labeled the Core in Fig. 2: it contains 44% of the files in the codebase. The next largest cyclic group is much smaller containing only 118 files. How this cluster is derived is explained in detail in [24]. In Table 2 all metrics are presented with their real numbers. Due to the nature of our data all variables (except the binary *CYCLIC* variable) are converted to their natural logarithms (LN) in calculating correlations (Table 4) and in our regression models (Table 5).

Our dependent variable (*VULN*, 1/0) has a value of one if the file in question was changed to fix a defect classified as a vulnerability, and zero otherwise.

In an attempt to untangle the coupling measures relation to vulnerabilities we have also used direct coupling within the main cyclic group (*DFIxC & DFOxC*) and outside of this group (*DFIxNoC & DFOxNoC*).

[2] https://www.chromium.org.

[3] https://silverthreadinc.com.

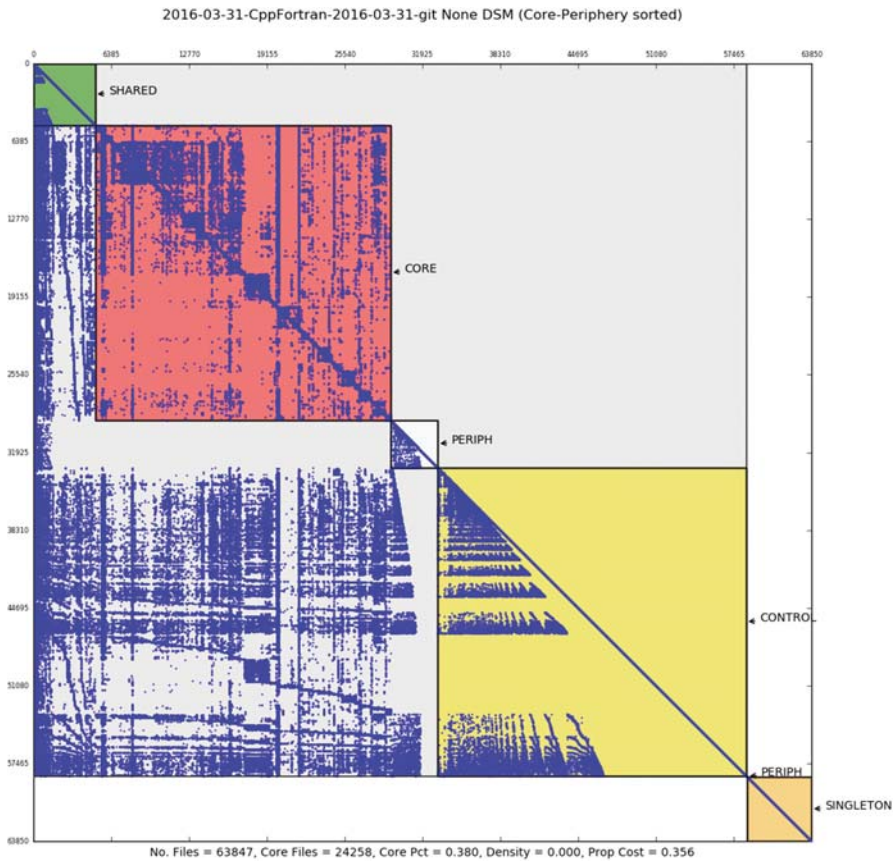

Fig. 2. A visualization of the Google Chrome 2016 software architecture showing all direct dependencies and sorted by different coupling categories. This figure shows the full set of files associated with Chrome, thus a larger number of files than what is used in our analysis. The Silverthread analysis tool produced the figure, see [6] for more information. The coupling categories shown in the figure are based on the number of indirect fan-in and fan-outs, where; the "Shared" group contains files that have high fan-in and low fan-out, "Core" is the largest cyclic group with both high fan-in and fan-out, "Peripheral" files have low indirect coupling, the "Control" group has low fan-in and high fan-out, and the "Singletons" have no coupling at all.

Table 2. Descriptive statistics for 2016 Google Chrome metrics.

2016/n = 16,268	Max	Min	Mean	Median	St. dev
VULN	1	0	0.02	0	0.13
SLOC	69,702	0	196.18	79	854.61
MCCABE	868	0	9.13	5	20.13
COMMR	39	0	0.33	0.17	0.88
CHURN	634,536	1	883.40	249	5,734
DFI	9,381	1	6.09	2	76.97
DFO	355	1	15.69	12	16.05
IndFI	49,570	1	22,996	23	24,705
IndFO	29,949	1	27,227	29,314	7,554
CYCLIC	1	0	0.44	0	0.50

4.2 Chrome Vulnerabilities

We collected the Google Chrome related vulnerabilities using the bug Tracker system[4] used by the developers in the Chromium project. In Tracker, bugs classified as vulnerabilities are registered with both their external CVE[5] ID and the internally used bug ID. (CVE, which stands for Common Vulnerabilities and Exposures, is a published list of security vulnerabilities that provides unique IDs for publicly known security issues. Some CVEs are associated with multiple internally defined bugs.)

We then used the internal bug IDs to track the files in the Chrome architecture that were changed in order to fix each vulnerability-classified bug. We did this by extracting the commits that specified fixing a vulnerability bug that was tagged with the internal bug ID.

As noted in [36] many CVEs are associated with external projects, thus not identifiable using the Chrome commits. Nguyen and Massacci claim that two thirds of the Chrome vulnerabilities are unverifiable due to this issue, which seems to be in line with our findings. We found 1,063 unique CVEs (on April 14th, 2016) associated with 1,070 bugs in the Chrome bug Tracker. Going through the commits gave us 407 bugs associated with 390 CVEs, which were fixed by patching 965 C- & header-files.

The architecture displayed in Fig. 2 contains the complete Google Chrome architecture as of March 31st, 2016, including both C-files and header files. These in turn were associated with 288 CVEs corresponding to 294 internal vulnerability bugs fixed by modifying 621 files. At this point, we excluded the header-files since these come hand in hand with the C-files and are thus associated with the same bugs and CVEs. In doing so we found only nine non-redundant CVEs and vulnerability bugs, and decreased the total number of files for analysis by 32,418 files.

The 2016 Chrome architecture contains a directory called "third_party/WebKit" which is a recently (2015) merged set of files from another vendor (Apple). From an architectural coupling perspective this directory, and its recent bulk merge, creates a

Table 3. CVE and Chrome file data leading to our analysis set of files.

	CVEs	Bugs	Vuln. files	Total files
Total contained in tracker	1,063	1,070	Not known	Not appl.
C- & h-files fixed by Google commits	390	407	965	Not appl.
Found for March 2016 architecture	288	294	621	63,847
Subtract:				
h-files	9	9	223	32,418
third_party/WebKit	94	95	108	3,015
Missing data	0	0	0	12,146
Final:				
Analysis sample	185	190	290	16,268

[4] http://bugs.chromium.org/p/chromium/issues/.

[5] https://cve.mitre.org/

situation where most of the directory is not a part of the overall architectural structure. It has its own special structure that is not representative of the rest of the system. To avoid mixing systems with different histories and structure, we excluded this directory as well. Ninety-four CVEs were associated with this directory. Finally, one of our key variables, code churn (CHURN), had missing data for many of the C-files. Dropping these files did not reduce the CVE count, but did reduce the number of files by 12,146 to 16,268.

In summary, our final dataset contains 185 unique CVEs associated with 190 vulnerability bugs fixed in patches of 290 C-files, and the total C-file set to compare with is 16,268. The numbers for this data cleanse are detailed in Table 3.

5 Chrome Metrics and Vulnerable Files

In this section we explore the relationship between software component metrics and different architecture coupling metrics in vulnerability-associated files of the Google Chrome architecture from 2016.

5.1 Findings

Table 4 presents a correlation matrix for the variables we study. As indicated, we have taken the natural logarithm of each variable except the binary variable *CYCLIC*. From the table, we can see that *CHURN* is highly correlated with source lines of code (*SLOC*), cyclomatic complexity (*MCCABE*), and direct fan-out (*DFO*). That is, files that are associated with many changes in general (excluding vulnerability bug changes) also have more source lines of code, higher cyclomatic complexity, and a higher number of direct fan-out dependencies. Further, we see that all of our software metrics, including the different types of coupling, are significantly correlated with vulnerability bug files (*VULN*). Namely, files that have been changed a lot, that have a low comment ratio, many source lines of code, high cyclomatic complexity, and high coupling are all associated with vulnerability bugs.

Unfortunately, due to the high correlation between variables we can't include them all in the same regression model. For the metrics code churn, source lines of code, cyclomatic complexity, and direct fan-out we basically need to choose one to include in the regression. Code churn has been proven before to be a good predictor (see e.g. [3]) and it also has the highest correlation with our *VULN* variable. Therefore, we chose to include *CHURN* in our regression models and not the others.

Model 1 in our regression contains three traditional software metrics used in defect and vulnerability prediction; code churn, comment ratio, and direct fan-in. This is a good starting model since these all have been used successfully before. As can be seen in Table 5 all three variables significantly contribute to our model. Since we want to explore coupling further, Model 2 tests whether indirect fan-in (*IndFI*) and indirect fan-out (*IndFO*) add any explanatory power over *DFI*. Model 3 looks at cyclic coupling (*CYCLIC*) instead of *IndFI* and *IndFO*. So far, we can see that the indirect coupling metrics perform better together with code churn and commenting ratio

Table 4. Correlation table for vulnerability bugs and complexity metrics in the 2016 Google Chrome software architecture.

	VULN	*SLOC*	*MCCABE*	*COMMR*	*CHURN*	*DFI*	*DFO*	*IndFI*	*IndFO*	*CYCLIC*	*DFIxC*	*DFOxC*	*DFIxNoC*	*DFOxNoC*
VULN	1													
SLOC	.135*	1												
MCCABE	.110*	**.804***	1											
COMMR	-.021*	-.299*	-.199*	1										
CHURN	.180*	**.801***	**.675***	-.103*	1									
DFI	.084*	.203*	.128*	-.062*	.195*	1								
DFO	.140*	**.707***	**.622***	-.248*	**.656***	.136*	1							
IndFI	.063*	.144*	.114*	-.022*	.134*	**.603***	.149*	1						
IndFO	.026*	.384*	.400*	-.226*	.277*	.043*	.577*	.048*	1					
CYCLIC	.066*	.202*	.188*	-.085*	.183*	.508*	.245*	**.929***	.243*	1				
DFIxC	.088*	.218*	.177*	-.073*	.219*	**.816***	.203*	**.718***	.188*	**.774***	1			
DFOxC	.115*	.338*	.296*	-.109*	.333*	.480*	.437*	**.859***	.225*	**.925***	**.725***	1		
DFIxNoC	-0.014	-.042*	-.094*	.023*	-.059*	.222*	-.125*	-.248*	-.248*	-.494*	-.382*	-.457*	1	
DFOxNoC	-.022*	.155*	.138*	-.065*	.123*	-.415*	.263*	**-.813***	.181*	**-.813***	**-.629***	**-.753***	.399**	1

* Correlation is significant at the 0.01 level, **bold** numbers indicate high correlation problematic for regression model

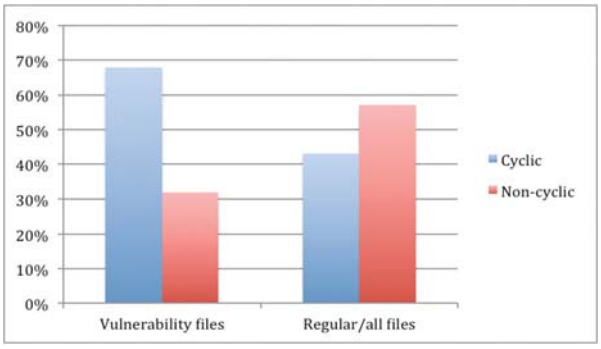

Fig. 3. Vulnerability associated files in the Cyclic and Non-cyclic groups.

compared to direct and cyclic coupling. This is not surprising since it is a more coarse grained metric that includes more information than both cyclic and direct coupling.

In Model 4 and 5 we test if direct fan-in and fan-out within the main cyclic group and outside this group adds any explanatory power. Again, due to the correlation among the measures, we have to divide our direct coupling measures from each other. DFI within the cyclic files (*DFIxCyc*) is tested in Model 4 and *DFO* within the cyclic files (*DFOxCyc*) in Model 5. Our best model seems to be the one including code churn, comment ratio, direct fan-out within the cyclic group (*DFOxCyc*) together with direct fan-in within the non-cyclic group (*DFIxNoCyc*), that is Model 5. Meaning that files that are cyclically coupled together in a large co-dependent network with a high degree of direct fan-out coupling and files that are not in this large cyclic group but have a high degree of direct fan-in coupling (the group in Fig. 2 called Shared), and with many changes in the past and a low commenting ratio – are more likely to contain vulnerabilities.

These results are in line with related work and this is where we are at with vulnerability prediction today.

Table 5. Binary logistic regression for vulnerability bug files in 2016 Chrome architecture.

VULN	Model 1	Model 2	Model 3	Model 4	Model 5
COMMR	−1.003**	−.880*	−.878*	−.993**	−.714*
CHURN	.861***	.874***	.876***	.859***	.805***
DFI	.215***				
IndFI		.063***			
IndFO		.004			
CYCLIC			.568***		
DFIxC				.216***	
DFOxC					.286***
DFIxNoC				.137	.381**
Constant	−9.683***	−9.999***	−9.869***	−9.647***	−9.710***
Chi-square	531.038***	538.206***	534.718***	531.613***	553.748***
Cox&Snell R^2	0.032	0.033	0.032	0.032	0.033
Nagelkerke R^2	0.197	0.199	0.198	0.197	0.205

n = 16,268, * $p < 0.05$, ** $p < 0.01$, and ***$p < 0.001$

5.2 Use of Different Sets of Data Available

As mentioned in the data section our dataset only represents a subset of the files in the 2016 March release of Google Chrome. We have run all analyses on the different subsets (cf. Table 3) as a robustness check, there are some variance but the main story looks similar.

One could argue that vulnerabilities from e.g. 2010 can't be analyzed based on an architecture from 2016. As a robustness check we have also looked at the 2010 architecture (the first stable release of Chrome). We also divided our vulnerability data in two equally large sets and looking at the older half of vulnerabilities for the 2010 architecture, and the younger half of vulnerabilities for the 2016 architecture.

The correlations and regressions with the 2010 and 2016 architecture and all vulnerability bugs, as well as the division of bugs between 2010 and 2016 look fairly similar. We did lose some power when dividing the bug dataset between the architectures, this had us chose the architecture associated with most vulnerabilities, that is the 2016 architecture and all vulnerability related bugs.

If we compare Table 2 (2016 Chrome stats.) and Table 6 (2010 Chrome stats.) we can see that some of the software metrics have changed, e.g. the comment ratio and direct fan-in and fan-out. Although the maximum values have significant increases, the differences when considering the means or medians are not that large. For cyclomatic complexity and code churn the means actually went down between 2010 and 2016. The main differences can be seen in the coupling measures for indirect fan-in and fan-out, where the means increased from 3,000–4,000 to 23,000–27,000. This is related to the considerable increase in size of the main cyclic group of dependent files, the "Core" in 2010 contained 4,049 files and in 2016 it had grown to 24,258 files.

Regarding the visualization of the architecture seen in Fig. 4 one can conclude that the 2010 and 2016 architectures look very similar in terms of coupling. For both architectures we have a large "Core" (that is, a large cyclic group where everyone depends on one another). In 2010 the Core was 33% of the architecture and in 2016 38%, thus even though this group of files grew considerably in number, it grew in proportion to the rest of the codebase. Both architectures also have large Control groups (files with high indirect fan-out and low indirect fan-in), and small groups of Shared files (files with high indirect fan-in and low indirect fan-out), Peripheral files (low indirect coupling), and Singletons (no coupling). This architectural similarity we interpret as an indication that our coupling measures are comparable with vulnerabilities for either architecture as long as the file associated with the vulnerability bug is present in the architecture.

6 Discussion and Future Work

Our work is the first study to use the Google Chromium project to explore the relationship between software metrics and vulnerabilities. As such it adds evidence to the total body of knowledge on this topic, which is still fairly unexplored (especially in comparison with the relationship between software metrics and generic defects).

The main weakness in our study we believe is that, because of their high level of collinearity, the different coupling measures have essentially equal predictive power in our regressions. The component-level metrics—source lines of code, code churn, cyclomatic complexity—are also highly correlated with each other and with direct fan-out. For this reason, we are unable (in this codebase) to tease apart the separate contribution of conceptually distinct, but empirically indistinguishable, types of complexity on vulnerability. Hopefully, we (or others) can explore these questions further using other software architectures in order to better understand the linkages between complexity, coupling and vulnerabilities.

We are not alone in having difficulty determining the relationship between software metrics and vulnerabilities. As reported in the section on related work, other studies also report weak power or non-statistically significant variables e.g. [1, 2]. Prediction models report either too many false positives or low precision. Thus this seems to be a generally difficult area to research.

Statistical problems, like collinearity, make it difficult to identify the causal mechanisms linking complexity and coupling with vulnerabilities. However, most studies are able to report valid correlations. Thus a growing body of collective evidence shows that variables like size, complexity, code churn, and coupling are associated with and thus likely to increase the incidence of vulnerabilities. The correlations reported in Table 4 and the regressions in Table 5 indicate a significant relation between vulnerable files, traditional software metrics and coupling measures, hence add to this mounting body of evidence. However, future work is needed.

Many studies present vulnerability prediction models with a large set of variables. However, very few describe the details of the variables included. What variables do actually contribute to the prediction model and which do not? This is especially interesting for architecture coupling and component complexity, since some of the

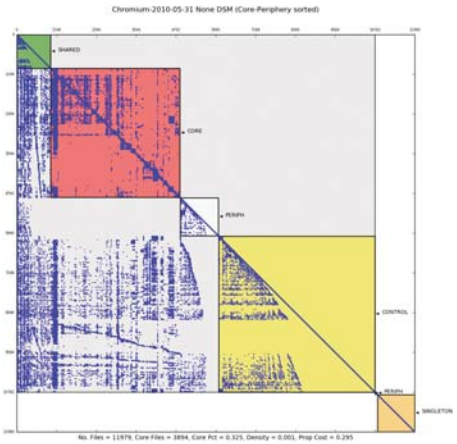

Fig. 4. Google Chrome 2010 software architecture showing a large cyclic cluster (here called the core) and other similar features as the 2016 version.

Table 6. Descriptive statistics for complexity metrics in Google Chrome 2010.

2010 / n = 6,333	Max	Min	Mean	Median	St.dev
VULN	1	0	0.02	0	0.15
SLOC	72,847	0	300.72	121	1,554.97
MCCABE	700	0	10.75	5	25.15
COMMR	163	0	0.44	0.23	2.24
CHURN	213,099	10	2,096.52	846	6,248.25
DFI	2,281	1	6.03	2	36.19
DFO	189	1	14.23	11	13.33
IndFI	8,649	1	3,204.69	9	4,146.90
IndFO	5,125	1	4,349.66	4,916	1,589.18
CYCLIC	1	0	0.36	0	0.48

papers we have studied report these as good predictors e.g. [3] while other say it shows weak or no relation to vulnerabilities e.g. [1, 2]. In general it would have been interesting to get this information from all studies, in e.g. the form of a correlation table or detailed prediction models.

As indicated, our findings on the relationship between of software metrics and vulnerabilities are mixed. Thus, future work is very much needed. First, we believe doing more studies in general on this topic is necessary. So far, there are few studies and thus few software systems have been investigated. For generalizability, more work is needed, including not only open source projects but also commercial software. Secondly, we have only found one study that compares general defects and vulnerabilities using the same data set. Camilo et al. [37] also studied the Google Chromium project and found "that bugs and vulnerabilities are empirically dissimilar groups, warranting the need for more research targeting vulnerabilities." More studies of this kind are necessary. Specifically we would like to be able to say what the main differences between a vulnerability bug and regular bug are in terms of complexity, size, code churn, commenting, and coupling. This could help us build more accurate prediction models.

7 Conclusions

Managing software vulnerabilities has become one of the top issues in today's society. Previous research on software defects, and to some extent vulnerabilities, showed that component level metrics (e.g. complexity and code churn) and architecture measures (e.g. coupling) can be good predictors of where problems are likely to occur. In this study we studied the Google Chromium project and found that all our metrics, both component and architecture level, are highly correlated with files that have been

patched in order to fix vulnerability classified bugs. We also set out to test whether different software architecture coupling measures were correlated with a higher incidence of vulnerabilities. In our tests we found it difficult to conclusively distinguish between our different measures of coupling, but the indication is that indirect coupling performs better than direct coupling, and the best model is a combination of cyclic coupling and direct coupling. We strongly believe that the indications in our study together with other related research show that software metrics of different kinds can be very helpful in locating vulnerabilities, but that more work is needed.

References

1. Shin, Y., Williams, L.: Is complexity really the enemy of software security?. In: Proceedings of the 4th ACM Workshop on Quality of Protection, pp. 47–50 (2008)
2. Morrison, P., Herzig, K., Murphy, B., Williams, L.: Challenges with applying vulnerability prediction models. In: Proceedings of the 2015 Symposium and Bootcamp on the Science of Security, p. 4 (2015)
3. Shin, Y., Meneely, A., Williams, L., Osborne, J.A.: Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE Trans. Software Eng. **37**(6), 772–787 (2011)
4. Walden, J., Stuckman, J., Scandariato, R.: Predicting vulnerable components: software metrics vs text mining. In: IEEE 25th International Symposium on Software Reliability Engineering, pp. 23–33 (2014)
5. Moshtari, S., Sami, A., Azimi, M.: Using complexity metrics to improve software security. Comput. Fraud Secur. **5**, 8–17 (2013)
6. MacCormack, A., Sturtevant, D.: Technical debt and system architecture: the impact of coupling on defect-related activity. J. Syst. Software **120**, 170–182 (2016)
7. Sturtevant, D.J.: System design and the cost of architectural complexity. Doctoral dissertation, Massachusetts Institute of Technology (MIT) (2013)
8. Akaikine, A.: The impact of software design structure on product maintenance costs and measurement of economic benefits of product design. Master thesis, Massachusetts Institute of Technology (MIT) (2010)
9. Catal, C., Diri, B.: A systematic review of software fault prediction studies. Expert Syst. Appl. **36**(4), 7346–7354 (2009)
10. Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S.: A systematic literature review on fault prediction performance in software engineering. IEEE Trans. Software Eng. **38**, 1276–1304 (2012)
11. Kitchenham, B., Pickard, L., Linkman, S.: An evaluation of some design metrics. Software Eng. J. **5**(1), 50–58 (1990)
12. Basili, V.R., Briand, L.C., Melo, W.L.: A validation of object-oriented design metrics as quality indicators. IEEE Trans. Software Eng. **22**, 751–761 (1990)
13. Nagappan, N., Ball, T.: Use of relative code churn measures to predict system defect density. In: Proceedings of the 27th International Conference on Software Engineering (ICSE), pp. 284–292 (2005)
14. Schröter, A., Zimmermann, T., Zeller, A.: Predicting component failures at design time. In: Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering, pp. 18–27 (2006)

15. Zimmermann, T., Nagappan, N.: Predicting defects using network analysis on dependency graphs. In: Proceedings of the 30th International Conference on Software Engineering (ICSE), pp. 531–540 (2008)
16. Steff, M., Russo, B.: Measuring architectural change for defect estimation and localization. In: Proceedings of the International Symposium on Empirical Software Engineering and Measurement, pp. 225–234 (2011)
17. Neuhaus, S., Zimmermann, T., Holler, C., Zeller, A.: Predicting vulnerable software components. In: ACM Conference on Computer and Communications Security (CCS), pp. 529–540 (2007)
18. Neuhaus, S., Zimmermann, T.: The beauty and the beast: vulnerabilities in red hat's packages. In: Proceedings of the Annual Technical Conference on USENIX, p. 30 (2009)
19. Zimmermann, T., Nagappan, N., Williams, L.: Searching for a needle in a haystack: predicting security vulnerabilities for windows vista. In: Proceedings of the International Conference on Software Testing, Verification & Validation, pp. 421–428 (2010)
20. Nguyen, V.H., Tran, L.M.: Predicting vulnerable software components with dependency graphs. In: Proceedings of the 6th International Workshop on Security Measurements and Metrics, pp. 3:1–3:8 (2010)
21. Chowdhury, I., Zulkernine, M.: Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. J. Syst. Archit. **57**(3), 294–313 (2011)
22. Hovsepyan, A., Scandariato, R., Steff, M., Joosen, W.: Design churn as predictor of vulnerabilities? Int. J. Secure Software Eng. **5**(3), 16–31 (2014)
23. MacCormack, A., Baldwin, C., Rusnak, J.: Exploring the duality between product and organizational architectures: a test of the "mirroring" hypothesis. Res. Policy **41**(8), 1309–1324 (2012)
24. Baldwin, C.A., MacCormack, A., Rusnak, J.: Hidden structure: using network methods to map system architecture. Res. Policy **43**(8), 1381–1397 (2014)
25. Heiser, F., Lagerström, R., Addibpour, M.: Revealing hidden structures in organizational transformation – a case study. In: Persson, A., Stirna, J. (eds.) CAiSE 2015. LNBIP, vol. 215, pp. 327–338. Springer, Cham (2015). doi:10.1007/978-3-319-19243-7_31
26. Lagerström, R., Addibpour, M., Heiser, F.: Product feature prioritization using the hidden structure method: a practical case at Ericsson. In: Proceedings of the Portland International Center for Management of Engineering and Technology (PICMET) Conference. IEEE, September 2016
27. Lagerström, R., Baldwin, C., MacCormack, A., Dreyfus, D.: Visualizing and measuring enterprise architecture: an exploratory biopharma case. In: Grabis, J., Kirikova, M., Zdravkovic, J., Stirna, J. (eds.) PoEM 2013. LNBIP, vol. 165, pp. 9–23. Springer, Heidelberg (2013). doi:10.1007/978-3-642-41641-5_2
28. Lagerström, R., Baldwin, C., MacCormack, A., Dreyfus, D.: Visualizing and measuring software portfolio architecture: a flexibility analysis. In: Proceedings of the 16th International DSM Conference (2014)
29. MacCormack, A., Lagerström, R., Dreyfus, D., Baldwin, C.: Building the agile enterprise: IT architecture, modularity and the cost of IT change. Harvard Business School Working Paper, No. 15-060, (2015) (revised August 2016)
30. Albrecht, A.J., Gaffney, J.E.: Software function, source lines of code, and development effort prediction: a software science validation. IEEE Trans. Software Eng. **6**, 639–648 (1983)
31. Kan, S.H.: Metrics and Models in Software Quality Engineering. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
32. McCabe, T.J.: A complexity measure. IEEE Trans. Software Eng. **4**, 308–320 (1976)
33. Aggarwal, K.K., Singh, Y., Chandra, P., Puri, M.: Sensitivity analysis of fuzzy and neural network models. ACM SIGSOFT Software Eng. Notes **30**(4), 1–4 (2005)

34. Simon, H.A.: The architecture of complexity. Proc. Am. Philos. Soc. **106**, 467–482 (1962)
35. Sosa, M., Mihm, J., Browning, T.: Linking cyclicality and product quality. Manufact. Serv. Oper. Manage. **15**(3), 473–491 (2013)
36. Nguyen, V.H., Massacci, F.: The (un) reliability of NVD vulnerable versions data: an empirical experiment on google chrome vulnerabilities. In: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, pp. 493–498. ACM (2013)
37. Camilo, F., Meneely, A., Nagappan, M.: Do bugs foreshadow vulnerabilities? A study of the chromium project. In: Proceedings of the 12th IEEE/ACM Working Conference on Mining Software Repositories (MSR), pp. 269–279. IEEE (2015)