


Do Bugs Propagate? An Empirical Analysis of Temporal Correlations Among Software Bugs

Xiaodong Gu ✉ 

School of Software, Shanghai Jiao Tong University, China

Yo-Sub Han ✉ 

Department of Computer Science, Yonsei University, Seoul, South Korea

Sunghun Kim ✉

Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong

Hongyu Zhang ✉

The University of New Castle, Australia

Abstract

The occurrences of bugs are not isolated events, rather they may interact, affect each other, and trigger other latent bugs. Identifying and understanding bug correlations could help developers localize bug origins, predict potential bugs, and design better architectures of software artifacts to prevent bug affection. Many studies in the defect prediction and fault localization literature implied the dependence and interactions between multiple bugs, but few of them explicitly investigate the correlations of bugs across time steps and how bugs affect each other. In this paper, we perform social network analysis on the temporal correlations between bugs across time steps on software artifact ties, i.e., software graphs. Adopted from the correlation analysis methodology in social networks, we construct software graphs of three artifact ties such as function calls and type hierarchy and then perform longitudinal logistic regressions of time-lag bug correlations on these graphs. Our experiments on four open-source projects suggest that bugs can propagate as observed on certain artifact tie graphs. Based on our findings, we propose a hybrid artifact tie graph, a synthesis of a few well-known software graphs, that exhibits a higher degree of bug propagation. Our findings shed light on research for better bug prediction and localization models and help developers to perform maintenance actions to prevent consequential bugs.

2012 ACM Subject Classification Software and its engineering → Maintaining software

Keywords and phrases empirical software engineering, bug propagation, software graph, bug correlation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.11

Acknowledgements The authors would like to thank anonymous reviewers for their very insightful comments and constructive suggestions in greatly improving the quality of this paper.

1 Introduction

Software bugs are not isolated [30, 62], rather they may interact, affect each other, and trigger consequential bugs over certain software artifacts [38, 54, 56]. Identifying and understanding bug correlations could help developers localize bug origins, predict future bugs, and design better architectures to prevent bug propagation. Therefore, the study of bug propagation is of tremendous importance from both analysis and design points of view.

There has been much work on bug co-occurrence and localities, indirectly implying the propagation of bugs [24, 30, 48]. For example, Kim et al. [30] found that bugs do not occur in isolation, but rather in bursts of several related bugs. Zimmermann et al. [62, 63] investigate bug correlations over software dependency graphs and utilize the correlation effect for defect prediction. While these studies have shown the evidence of spatial correlation among bugs,



© Xiaodong Gu, Yo-Sub Han, Sunghun Kim, and Hongyu Zhang;
licensed under Creative Commons License CC-BY 4.0

35th European Conference on Object-Oriented Programming (ECOOP 2021).

Editors: Manu Sridharan and Anders Møller; Article No. 11; pp. 11:1–11:21

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

11:2 Do Bugs Propagate?

they do not investigate the existence of bug correlation across time steps. There are many unanswered questions about temporal correlations among the bugs. More specifically: 1) **Do software bugs propagate over time?** 2) **How do they propagate?** and 3) **Why do they propagate?**

In this paper, we investigate the temporal (a.k.a., time-lag) correlations among software bugs along with software evolution. We adopt the correlation analysis methodology from social science, which has shown effectiveness in identifying propagation of social phenomena (such as obesity and suicide) [12, 19]. Like in social network analysis [12], we define temporal correlation as the correlation between bugs across time steps.

In social science, corrections of social phenomena depends on social ties such as parents, relatives and friends. These social ties are usually represented as graphs (i.e., friend graph, neighbor graph, and sibling graph) [12, 50, 51]. Researchers then perform correlation analysis on these graphs [19, 12] to analyze the temporal correlation phenomenon. Choosing different graphs may lead to different findings. For example, obesity propagates more significantly on friend graphs than on sibling graphs [12]. Similar to social ties, there are also a various kinds of ties among software modules such as dependencies [41, 62, 63], collaborations [25, 45], and interactions [45]. We call such ties *software ties* which can be modeled by software graphs. For example, Zimmermann [62, 63] has built the dependency graph which characterizes data and call dependence between software artifacts and shows effectiveness in defect prediction. Pinzger et al. [45] proposed the developer-module network which models the relationships between developers and their contributions.

To explore the temporal correlation of software bugs on different software ties, we build graphs of three known artifact ties from the literature such as function calls [7], module inheritance [22], and file co-change [14, 24]. Nodes in each graph stand for source code files, while edges refer to software ties such as calling, inheritance, and co-change (CC) relationships. We build software tie graphs for four open-source projects (HTTPClient, Jackrabbit, Lucene, and Rhino) and then perform the longitudinal logistic regression [12] on the bug and metric statistics on these graphs. We measure the temporal correlation among bugs using the coefficients of the longitudinal logistic regression (LRC) and compare the results with well-recognized propagation phenomena such as obesity [12] and happiness [19].

Our experimental results show that there are significant temporal correlations among bugs through the three basic software ties. The logistic regression coefficients (LRC) of the three basic graphs are 0.19, 0.32 and 0.39, respectively, showing far more compelling results than the random graph (LRC=0) and are generally more significant than social events such as happiness (LRC=0.12) and obesity (LRC=0.28) [12, 19]. This answers our questions raised in the beginning: **bugs can propagate along with software evolution, through common software ties, and the propagation can be significant.** Having had more buggy neighbors among these ties in the past increases the likelihood of being buggy in the near future.

Based on the experimental results, we propose a hybrid graph (tie) which synthesizes basic graphs by merging and selecting their edges according to their temporal correlation. The proposed hybrid graph exhibits a higher degree of temporal correlation than basic graphs, with an average LRC of 0.66, which is greater than the values of basic graphs. That means, the hybrid graph provides a more efficient way for identifying temporal correlations among bugs.

The properties of temporal bug correlation have many benefits for developers. For example, using the correlation property, developers can link related bug reports, predict potential consequential bugs, and perform maintenance actions to prevent bug propagation.

Developers can also improve the quality of software by designing better artifact structures which have less bug correlation. The software ties between temporal correlated bugs can facilitate the localizing of correlated bugs and the understanding of their root causes.

Overall, our study makes the following contributions:

- We evaluate the temporal correlation of software bugs on three software ties (graphs), and show that software bugs are indeed temporally correlated over the ties. To our best knowledge, it is the first work that investigates the phenomenon of temporal correlation among software bugs.
- We design a hybrid graph and show that the graph highlights the bug correlation effects more significantly.
- We discuss reasons to the temporal bug correlations and suggest possible applications of our findings.

The rest of this paper is organized as follows. Section 2 discusses the background. Section 4 presents the common setup for bug correlation analysis. Section 5 presents the results for basic graphs, hybrid graph as well as the reasons of temporal bug correlation, followed by discussions in Section 6. Section 7 presents the related work, and Section 8 concludes the paper.

2 Background

This section introduces the background of *temporal correlation analysis* including the concepts, measures, and applications.

2.1 Temporal Correlation Analysis in Social Science

The temporal correlation analysis aims at identifying the consequential occurrence of a phenomenon in social science [12, 4]. For example, Christakis and Fowler [12] found that obesity can be “contagious” among various social ties such as friends and siblings. They evaluated the spread of the body-mass index in a densely interconnected social network of 12,067 people throughout a 32 year period and found weight gain in one person was associated with weight gain in his or her friends, siblings, spouses, and neighbors. Being overweight is also identified to be correlated among friends in schools [53]. Researchers estimated peer effects for adolescent weight using data from the *National Longitudinal Study of Adolescent Health*, and found that mean peer weight is correlated with adolescent weight especially among females.

Correlation analysis has also shown to be effective in studying online social networks such as Facebook, MySpace and Flickr [51]. For example, Anagnostopoulos et al. [4] identified social influence by investigating correlations between individuals in an online social network, and proposed measures to assess the correlations. Chenhao [50] measured the individual influence among online friends. They designed a social action tracking algorithm to assess the quantitative values of influence and utilized them to help predict users’ future actions. Kempe [29] considered temporal correlation as a way to choose the most influential sets that can maximize the spread of information.

2.2 Modeling Ties Using Graphs

Graphs, as desirable models to capture ties, are widely used for temporal correlation analysis [12, 50, 51]. We list several graphs used in different research areas in Table 1. In online social networks such as Flickr, users are denoted as nodes while ties such as friendships and

■ **Table 1** Graphs models for correlation studies in other areas.

Area	Graph	Node	Edge	Weight
social science [12]	obesity	individual	friend	–
social science [19]	happiness	individual	friend	–
data mining [4]	Flickr	user	friend	influence

corporations are represented as edges [4]. In the social science literature, subject persons are denoted as nodes (egos and alters), while social ties such as friends, spouses, and siblings are represented as edges.

Identifying appropriate ties is critical in temporal correlation analysis. Different social ties correspond to different graphs and show different degrees of correlation, affecting the results of correlation analysis. For example, obesity exists in friend graphs but not in neighbor graphs [12]. Happiness is temporally correlated among neighbors but not among co-workers [19].

2.3 Measuring Propagation

Most propagation studies identify the propagation of a behavior by measuring temporal correlations between neighbors and egos in the tie graph. Intuitively, if a behavior propagates from one node to another, then the neighbor (the affected node) can also influence the ego (the source node). In that sense, the behavior occurrence between the neighbor and the ego can have a high correlation [4, 6]. In other words, propagation exists only if there exist correlations among egos and neighbors.

Such temporal correlation can be identified using the longitudinal logistic regression (LLR) [9, 12]. The LLR is a logistic regression model for longitudinal data (i.e., data that tracks the same subjects at multiple points in time). It assumes that the ego's behavior status at any given time t is a function of various attributes, including intrinsic features of the ego (e.g., age, sex and education level), any previous behavior of the ego at time $t-1$ and neighbors' disease status at times $t-1$ and t . Let n^{t-1} denote the number of neighbors that are affected at time $t-1$, y^t denote the ego's affection status (1=affected, 0=otherwise) at current time t , and x_i^t ($i=1\dots N$) denote the ego's intrinsic metrics at time t . The probability that the ego is affected at present $p(y^t)$ can be estimated as:

$$\ln\left(\frac{p(y^t)}{1-p(y^t)}\right) = \alpha n^{t-1} + \beta_0 + \beta_1 x_1^t + \dots + \beta_N x_N^t \quad (1)$$

where y^t stands for the dependent variable; n^{t-1} and all x^t 's are independent variables. α and β represent the regression coefficients for each independent variable [4].

The longitudinal regression model is commonly solved by the generalized estimating equation (GEE) [34, 46] which accounts for multiple observations of the same ego across examinations and across ego-neighbor pairs. By running GEE on the LLR model, we obtain the coefficients (e.g., α, β) for Equation 1. The temporal correlation is usually measured by the first coefficient α which indicates the correlation between y^t and n^{t-1} in Equation 1. We denote α as LRC (logistic regression coefficient) [9, 12]. The higher the LRC, the more significant correlation between the two variables. LRC is a widely used measure for temporal correlations between social behaviors [19]. For example, happiness is shown to propagate among social ties with an LRC of 0.12 [19], while obesity among social ties shows an LRC of 0.28 [12]. In a random network where behaviors are independently generated, the average LRC is expected to be 0.

■ **Table 2** Overview of software tie graphs.

Graph	Node	Edge	Weight
function call graph	source files	function call	# call sites
type hierarchy graph	source files	inheritance	–
co-change graph	source files	co-change	# co-changes

It is worth noting that in some areas, having correlations between behaviors does not necessarily indicate the existence of propagation [12]. The correlations could come from homophily or confounding factors (i.e., two nodes share the same characteristics) [4, 6, 12]. For example, in the obesity studies, people may tend to make friends with others who have the similar weights. This suggests the need to distinguish the homophily and confounding factors from correlations according to different scenarios. One commonly used method to address this issue is the edge-reversal test [4, 12]. This test first reverses the direction of all edges and then run the longitudinal regression on the reversed graph. Since the homophily and confounding factors are only based on the fact that two nodes often share the same characteristics. These factors must be independent of the two individuals' identification of the other as a neighbor. Therefore reversing the edges will not change the behavior correlation estimate significantly [4]. In other words, if correlations change significantly by reversing the graph, we can exclude the homophily and confounding factors.

3 Study Methodologies

This section presents our study methodologies for identifying the temporal correlations among bugs. We first introduce the software tie graphs where temporal correlations are dependent on, followed by the measurement of temporal correlations.

3.1 Software Tie Graphs

Similar to common social ties such as friends, siblings, and spouses (Section 2.2), software artifacts also have a various kinds of ties such as dependencies [41, 62, 63], collaborations [25, 45], and interactions [45]. Graphs, as effective models of social ties, are also widely used to represent software artifacts' ties. For example, Zimmermann et al. [62, 63] represented dependencies between software binaries by introducing the dependency graph. They further observed a substantial correlation between central binaries and defects on the graph. Pinzger et al. [45] leveraged graphs to model developer-modules and found correlations between centrality measures of developer-module networks and failure-prone modules.

We define a *tie graph* as a directed graph $G = (V, E, w)$ where V denotes the set of nodes (source files), $E \subseteq V \times V$ is the set of edges (artifact ties), and $w : E \rightarrow \mathbb{R}$ represent the weighting function for each edge. We select three software artifact ties according to well-known defect factors in the software engineering literature [15, 21, 32, 36, 45, 62, 63], and build three corresponding graphs: *function call graph*, *type hierarchy graph*, and *co-change graph*. An overview of all software artifact tie graphs is presented in Table 2.

3.1.1 Function Call Graph

The function call graph models the caller-callee relationship between artifacts. Intuitively, bugs from the callee could affect the caller since the caller reuse its codes directly. If a function in file A calls another function in file B, A and B are represented as two nodes in

the graph, and a directed edge from node A to node B is added. The weight of each edge is the number of call sites between nodes.

Figure 1 (a) and (c) illustrate an example of call graph. The class *Signature* has a function *sampleMethod* which calls a function in object of another class *JavaTextLabelProvider*. Then, there are two nodes *Signature.java* and *JavaTextLabelProvider.java* in the call graph and an edge from node *Signature.java* to the node *JavaTextLabelProvider.java*. The weights (numbers in the edges) indicate the total numbers of call sites from a file to another file.

3.1.2 Type Hierarchy Graph

The type hierarchy graph captures the inherit relationships among entities. Type hierarchy indicates the inheritances and implementations among classes. It is not only an important representation of software design, but also reveals substantial dependencies among artifacts. Nodes in type hierarchy graphs also stand for files. If a class in file A extends a class in file B, a directed edge from A to B is added. The weight of each edge is 1 since there is no multiple inheritances between nodes. Figure 1 (a) and (d) show an example of type hierarchy graph. The class *Signature* extends another class *AbstractSignature*. Then, there are two nodes *Signature.java* and *AbstractSignature.java* in the hierarchy graph and an edge from node *Signature.java* to the node *AbstractSignature.java*.

3.1.3 Co-Change Graph

We also use co-change graphs to capture the co-change relationships among files. Files that were changed together have implicit couplings [14, 24]. Co-changed files could have relevant functionality, or be maintained by the same developer. In a co-change graph, nodes stand for source files. If two files were changed at the same time, an edge weighted with co-changed times will be connected to both files.

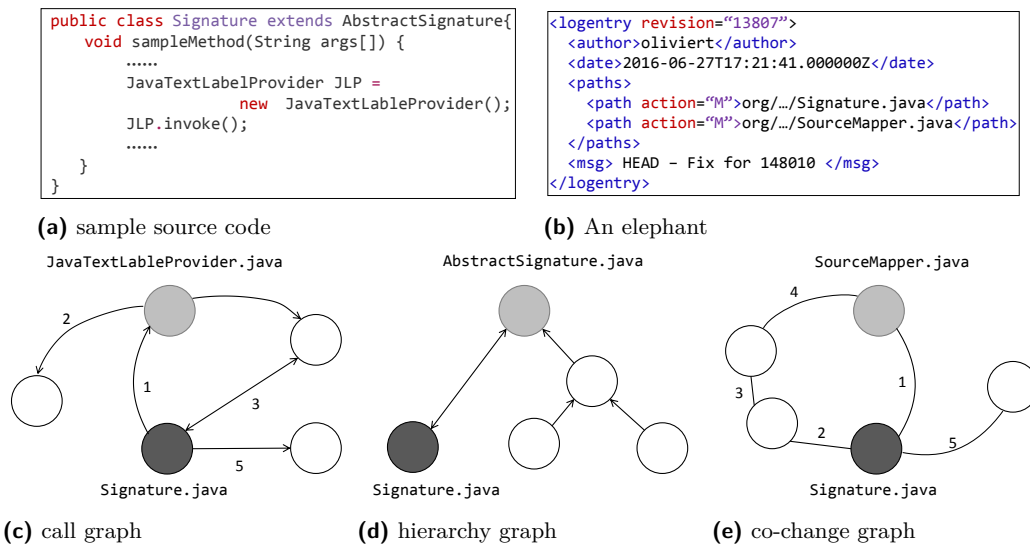
Figure 1 (b) and (e) illustrate an example of co-change graph. The file *Signature.java* was changed together with the file *SourceMapper.java*, therefore, there is an undirected edge from node *Signature.java* to node *SourceMapper.java*. The weights (numbers in the edges) indicate the total numbers of changes that two files were changed together.

3.2 Measuring Temporal Bug Correlations

We follow the longitudinal logistic regression methodology introduced in Section 3, and measure bug propagation by estimating the time-lag bug correlations between neighbors and egos across each consecutive pair of examination points (i.e., time $t-1$ and t). Specifically, we perform longitudinal logistic regressions (LLR) (Section 3) on the bug occurrence data and measure the time-lag bug correlations using the regression coefficient (LRC) (Section 2.3). Let $y^t = \{y_l^t\}_{l=1}^N$ denote the status of bug for N data instances at examination point t where $y_l^t \in \{1, 0\}$ (1=buggy, 0=clean). Let $x^t = \{x_l^t\}_{l=1}^N$ denote the corresponding independent variables, where $x_l^t = \{x_{1l}^t, x_{2l}^t, \dots, x_{Ml}^t\} \in R^M$ is a collection of common metrics that affect the presence of bugs such as lines of codes (LOC) and cyclomatic complexity (CYC) for instance l at examination point t . Let $n^t = \{n_l^t\}_{l=1}^N$ denote the number of buggy neighbors for each instance l at examination point t . The LLR model fits a logistic regression form [4]:

$$p(y_l^t = 1) = \frac{e^{\alpha n_l^{t-1} + \beta_0 + \beta_1 x_{1l}^t + \dots + \beta_M x_{Ml}^t}}{1 + e^{\alpha n_l^{t-1} + \beta_0 + \beta_1 x_{1l}^t + \dots + \beta_M x_{Ml}^t}} \quad (2)$$

where y_l^t represents the dependent variable, n_l^{t-1} and all x_l^t 's denote independent variables, and M stands for the number of independent variables for controlling. All independent



■ **Figure 1** Examples of source files and artifact tie graphs.

variables we select are shown in Table 3. We collect source code metrics (i.e., LOC, CYS, and ESS) using the Understand tool [1] and changing metrics (i.e., NOA and NOC) from the SVN commit logs.

4 Experimental Setup

This section describes the experimental setup. We first present our research questions. Then, we introduce the data sets, implementations, as well as baseline graphs.

4.1 Research Questions

We design our experiments to address the following research questions:

- **RQ1: (Temporal bug correlations on known software ties) Do software bugs propagate through the known software ties?** We build graph models for known software artifact ties such as call, inheritance, and co-changing. They are commonly used graphs to capture artifact dependencies [41, 62, 63] and collaborations [25, 45]. We perform longitudinal logistic regressions to measure the time-lag correlations (Section 2.3) between artifact ties. Then, we compare the results (LRC) with well-recognized correlation results in other research areas.
- **RQ2: (Temporal correlations on proposed graph) Do software bugs propagate across the proposed hybrid graph?** Based on the observation of basic graphs, we propose a hybrid graph which synthesizes basic graphs according to their correlation properties. We measure bug propagation across the proposed hybrid graph.
- **RQ3: (Casual mechanisms) What are the reasons of the temporal bug correlation phenomenon?** To explain the causal mechanism of such temporal correlations, we conduct a qualitative analysis in real software development and show a few reasons.

■ **Table 3** A taxonomy of the independent variables.

	metric	description	rationale
controls	LOC [32]	Lines of code	Large components are more likely to be defect-prone [32].
	CYC [36]	Sum of cyclomatic complexity of all nested functions or methods	More complex components are likely more defect-prone [36].
	ESS [36]	Sum of essential complexity for all nested functions or methods	
	NOA [21]	Number of authors	Components with many unique authors likely lack strong ownership, which in turn may lead to more defects [21].
	NOC [21]	Number of changes	The number of changes to code in the past was a successful predictor of faults [21].
	y_l^{t-1}	Number of defects in prior examination for instance l .	Defects may linger in components that were recently defective [21].
test	n_l^{t-1}	Number of neighbor bugs previously for instance l	Our hypothesis

■ **Table 4** Summary of subject projects.

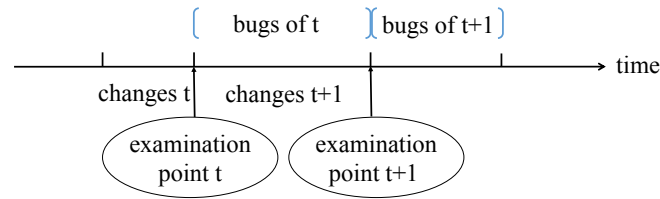
dataset	HTTPClient	JCR	Lucene	Rhino
time span	2007.4–2011.12	2006.12–2011.12	2010.3–2011.12	1999.4–2012.6
# unique files	281	1134	511	388
# total bugs	576	938	696	302
exam. points (selected releases)	4.0, 4.0.1, 4.1 α 2, 4.0.2, 4.1.1, 4.1.2, 4.2 α 1	1.1.1, 1.2.1, 1.3, 1.3.3, 1.4, 1.4.3, 1.4.6, 1.5, 1.5.5, 1.6, 2.0, 2.1, 2.2, 2.2.5, 2.2.7, 2.3, 2.3.3	3.0.1, 3.0.2, 3.0.3, 3.1, 3.2, 3.3, 3.4, 3.5	1_4r3, 1_5r1, 1_5r2, 1_5r4, 1_5r5, 1_6r1, 1_6r6, 1_7r2, 1_7r3, 1_7r4

4.2 Subjects and Data Collection

We collect bug repositories based on Herzig et al.’s manually verified bug database [23]. We select four projects¹ from their data sets: HTTPClient, Jackrabbit (JCR), Lucene, and Rhino. They are widely used in the software engineering literature, and more importantly the data sets have been manually verified [27]. Table 4 shows an overview of these data sets. They have various project periods (from 2 to 13 years) and different scales and bug densities. The HTTPClient contains only 281 distinct files with 576 bugs in total while Jackrabbit contains 1134 files with less than 1000 bugs.

Inspired by the study of obesity propagation [12], we separate the whole evolution of a project (i.e., all revisions) into different time periods by selecting several revisions as examination points ($t = 1, 2, \dots, T$, where T is the total number of examination points) (see Figure 2).

¹ For one of the five bug datasets, i.e., Tomcat, the source code of corresponding releases is not available.



■ **Figure 2** Illustration of examination points.

To make the examination points more reasonable, we select revisions when a release was published and includes bug fixes according to the release history. Different releases may have different purposes. For example, HTTPClient 4.2 is just released for enhancement while HTTPClient 4.1.3 is for bug fix. We cannot identify bugs during periods that contains no bug fix assignments. Therefore, in this example, we select as examination point the revision when release 4.1.3 was published. In order to balance the intervals between examination points, we also remove examination points that are too close to another (i.e., less than one month). We list all the selected examination points in Table 4.

At each examination point t , we collect JAR files of the corresponding release and capture a snapshot call graph and a hierarchy graph based on the JAR files. We generate all call and hierarchy graphs using the WALA tool (1.4.3) [2] with the default options.

We collect co-change information by parsing the commit history. For each examination revision t , we collect co-changed files in the previous examination periods ($t-1$ to t). If two files were committed at the same time during that period, we connect them with an edge in the corresponding co-change graphs.

Then, we label files at an examination point t by collecting bugs fixed at interval (t to $t+1$). For each bug in a data set, we compare the bug id against the messages in SVN commit logs to get the link between the bug and its fix revision. Then, we link the bug to files committed at that revision. For each file at each examination point (at time t) we count the numbers of fixed bugs during the examination period (t to $t+1$). If the number is more than zero, we label the file as buggy.

4.3 Implementation Details

We extract artifact metrics such as LOC and CYC using the Understand code analysis toolkit [1]. We implemented the LLR model using Matlab. The LLR Model is then estimated by solving a general estimating equation (GEE) with an equicorrelated working covariance structure [34]. We perform GEE using the GEEBOX tool [46] in Matlab.

4.4 Comparison

We compare the temporal correlation of software bugs in software ties with widely-accepted results in other research areas. We first compare the correlation value (LRC) to a randomly constructed graph where bugs are evenly distributed to the nodes. By mathematical definition [10], the LRC in a random graph is 0.0. $LRC = c$ ($c > 0$) means that the log odds of being buggy at time t increase by c times of the number of buggy neighbors increases at time $t-1$. For more references, we also compare bug correlation with correlation results from the social science literature such as obesity and happiness. Obesity is temporally correlated with the numbers of opposite sex siblings who are obese ($LRC=0.28$) [12] and happiness is temporally correlated with the numbers of neighbors who are happy ($LRC=0.12$) [19].

11:10 Do Bugs Propagate?

■ **Table 5** Results of time-lag bug correlation (‘.’ stands for results with p -values ≥ 0.05).

graphs	dataset	independent variables						
		n^{t-1} (LRC)	y^{t-1}	loc	cyc	ess	noa	noc
function call	HTTP	0.38
	JCR	0.13	0.01	0.01	-0.01	0.01	-0.07	0.20
	Lucene	0.10	.	0.00	.	0.01	0.26	.
	Rhino	0.16	.	0.01	-0.02	.	.	0.08
	Avg	0.19	0.01	0.01	-0.02	0.01	0.09	0.14
type hierarchy	HTTP
	JCR	0.23	0.004	0.01	-0.01	.	-0.05	0.17
	Lucene	0.41	.	.	.	0.01	0.27	0.06
	Rhino	.	.	0.004	-0.01	.	.	0.07
	Avg	0.32	0.004	0.01	-0.01	0.01	0.11	0.1
co-change	HTTP	0.51
	JCR	.	.	0.004	.	.	.	0.19
	Lucene	0.30	.	.	.	0.01	0.44	-0.19
	Rhino	0.36	-1.56
	Avg	0.39	-1.56	0.004	.	0.01	0.44	0

5 Results

This section presents our experimental results addressing the research questions (Section 4.1).

5.1 Bug Correlation on Known Software Ties (RQ1)

Table 5 shows the results of the logistic regression coefficients on known software artifact ties. Values in each row are the coefficients for different independent variables. We note ‘.’ for the values without statistical significance ($p \geq 0.05$), since they are meaningless [13]. The first column of these coefficients is the LRCs. The bold values indicate LRCs that are higher than that of the baselines. For example, the LRC for Rhino in the co-change graph is 0.36. That means, having one more buggy neighbor at time $t-1$, the ego’s risk (i.e., log odds) of being buggy at time t is 0.36 much higher.

As the results indicate, the call, hierarchy and co-change graphs have high positive LRCs in most subjects. The average LRCs for them are 0.19, 0.32, and 0.39, respectively. These values are much greater than the random graph (0.0) and are comparable to typical social correlations (obesity 0.28 and happiness 0.12). That means that temporal bug correlation is present on these graphs.

In summary, our results show that on some known ties (calling, extending and co-changing), software bugs show evidence of temporal correlation. There are correlations between a bug in one time period and bugs in the neighbors in the next time period.

Software bugs are temporally correlated in some known ties such as call, type hierarchy and co-changes.

5.2 Bug Correlation on Hybrid Tie (RQ2)

As the results in Section 5.1 indicate, different graphs may have different degree of bug correlation. We were wondering it is possible to synthesize a hybrid graph (software tie) that exhibits higher degree of correlation. Then, with the proposed hybrid graph, developers could identify subsequent bugs more efficiently. We propose to build a hybrid graph based

■ **Table 6** Results for the edge-reversal test (‘-’ stands for results with p-values ≥ 0.05 ; ‘*’ means that the result shows a significant change after reversing edges.).

graphs	dataset	independent variables						
		n^{t-1} (LRC)	y^{t-1}	loc	noc	noa	ess	cyc
function call	HTTP	0.38	-	-	-	-	-	-
	JCR	0.06(*)	0.01	0.01	0.21	-0.06	0.01	-0.01
	Lucene	0.16	0.02	-	-	0.26	0.01	-
	Rhino	-(*)	-	0.01	0.08	-	-	-0.02
type hierarchy	HTTP	-	-	-	-	-	-	-
	JCR	-(*)	0.004	0.01	0.19	-	0.01	-0.01
	Lucene	-(*)	-	-	0.08	0.27	0.01	-
	Rhino	-	-	0.004	0.07	-	-	-0.01
hybrid	HTTP	0.61(*)	-	-	-	-	-	-
	JCR	-(*)	0.004	0.01	0.19	-	0.01	-0.01
	Lucene	-(*)	-	-	-	0.29	0.01	-
	Rhino	0.44	-0.5	0.01	0.08	-	-	-

on bug correlation on basic graphs and evaluate temporal bug correlations on the proposed graph using approaches in Section 3.2.

5.2.1 Synthesizing Hybrid Correlation Graphs

Similar to the process of machine learning, we split all releases of a project into a training and a test set. We estimate LRCs for basic graphs in the training set. Then, in the test set, we synthesize a hybrid graph for each release based on the new basic graphs and their historical LRCs (i.e., LRCs in the training set).

We use a “greedy expansion” strategy when synthesizing the hybrid graph. We initially select the “best” graph (with the highest LRC) from basic graphs. Then, we “optimize” edges in the selected graph as follows: for edges in the “best” graph, if an edge appears in more than one basic graphs, we keep it in the hybrid graph. For edges in other basic graphs, if an edge appears in more than one basic graph, and the total LRC of other basic graphs is greater than that of the selected “best” graph, we add this edge in the hybrid graph as well.

Algorithm 1 illustrates the pipeline of constructing the hybrid graph. In the training stage, we compute LRCs for basic graphs as described in Section 3. In the test phase, we initially assign weights to edges in the new basic graphs using their historical LRCs (i.e., LRCs in the training set). Then, we merge their edges and add up the corresponding edge weights. Finally, we remove edges whose weights are no more than the maximum LRC in the merged graph. The remaining merged graph is then returned as the hybrid graph.

Figure 3 shows an example of the synthesis of hybrid graph. The top three graphs are basic graphs constructed from a project with historical LRCs of 0.6, 0.1, and 0.3, respectively. We also assign edge weights as 0.6, 0.1, and 0.3. To synthesize the hybrid graph, we first merge the edges together and add up the corresponding edge weights. Then, we select edges from the merged graph (bottom left) whose weights are greater than 0.6 (i.e., 0.7 and 0.9). Finally, these selected edges as well as the original nodes constitute the hybrid graph (bottom right).

² The *c*, *h*, and *cc* are short for call graph, hierarchy graph, and co-change graph, respectively

11:12 Do Bugs Propagate?

■ **Algorithm 1** Constructing hybrid bug correlation graph.

Input:

Basic Graphs, $G^i = (E^i, V, w^i)$, ($i \in \{c, h, cc\}$)²
Historical LRCs for basic graphs, $LRC(G^i)$

Output: Hybrid Graph, $G^* = (E^*, V, w^*)$

1: Choose the maximum LRC from all G^i

$$LRC_{max} = \max_{G^i \in \{G^c, G^h, G^{cc}\}} LRC(G^i); \quad (3)$$

2: $E^* = E^c \cup E^h \cup E^{cc}$

3: **for all** $(u, v) \in E^*$ **do**

4: $w^*(u, v) = w^c(u, v) + w^h(u, v) + w^{cc}(u, v)$

5: **if** $w^*(u, v) \leq LRC_{max}$ **then**

6: $E^* = E^* \setminus (u, v)$

7: **end if**

8: **end for**

9: $G^* = (E^*, V, w^*)$;

10: **return** G^*

5.2.2 LRC on The Hybrid Graph

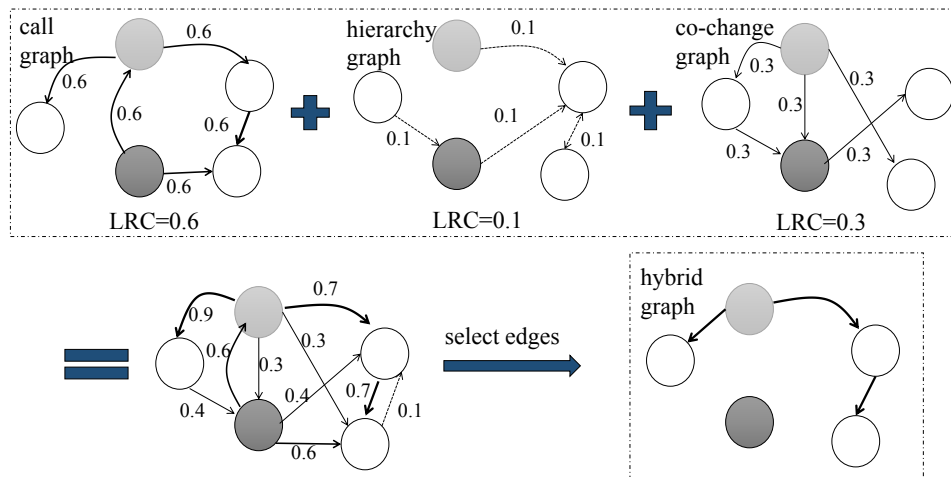
After constructing the hybrid graph for each release, we estimate temporal bug correlations on the proposed graphs using the same methodology (Section 3.2).

Table 7 shows LRCs for hybrid graphs. For each project, we set the first third of releases for training and the remaining releases for test. Each row shows LRCs of a specific graph for different datasets. The last column shows the average LRCs (excluding LRCs with $p > 0.05$) for each graph.

As shown in the results, the hybrid graph exhibits much higher LRCs in most subjects (1.09 in HTTPClient, 0.65 in Lucene, and 0.52 in Rhino). The average LRC for the hybrid graph is 0.66, much higher than those of call, hierarchy, and co-change graphs (0.18, 0.38 and 0.35, respectively). Table 6 shows the results for edge-reversal test. We reverse all the edges and present the coefficients again. Each line represents coefficients in terms of different independent variables. The first column shows the new LRCs for hybrid graph. The results show that after reversing all the edges, the LRCs change significantly in three out of four subjects.

The hybrid graph in Jackrabbit shows a maximized yet not improved LRC. We find it is constituted by the call graph and the hierarchy graph since the co-change graph in this subject shows no significant bug correlation. Then, since hierarchy graphs are relatively sparse and may have fewer intersections with call graphs. The resulting hybrid graph can be very sparse. On the other hand, both the call and the hierarchy graphs represent source code based dependencies. Their small intersections may have no information gain without combining other category of graphs such as co-change graphs. These could be the main reasons for such an outlier.

In summary, the degree of temporal bug correlation is much higher in the proposed hybrid graph than that in other basic software ties. The results indicate that the proposed hybrid graph can exhibit bug propagation more effectively.



■ **Figure 3** Hybrid graph construction.

■ **Table 7** Results of temporal bug correlation in the proposed hybrid graph. The bold numbers mean higher degree of correlation than basic graphs.

Graph	HttpClient	JCR	Lucene	Rhino	Avg
call	.	0.15	0.13	0.27	0.18
hierarchy	.	0.36	0.39	.	0.38
co-change	0.45	.	0.30	0.29	0.35
hybrid	1.09	0.36	0.65	0.52	0.66

The proposed hybrid graph provides a synthesized software tie that exhibits a higher degree of temporal bug correlation.

5.3 Casual Mechanisms behind the Temporal Bug Correlation (RQ3)

In this section, we explain the casual mechanisms behind temporal bug correlation through case studies.

The most fundamental and challenging question here is why bugs propagate. Bugs seem to be statically created at different times. How do they correlated between different artifacts?

To explain the causal mechanism of such time-lag correlations, we conduct a qualitative analysis in real software development. Specifically, we select some real bugs from our data set and analyze how bugs of one code can “propagate” to another piece of code. According to the bug reports and issue discussions, we identified three mechanisms for bug correlations.

1) Code Reuse: A possible causal mechanism for the temporal correlation could be the code reuse [47] (e.g., clone, inheritance, components, template, framework) on dependent artifacts. Dependent artifacts such as method calling and inheritance have high couplings [17]. They may reuse the codes or logics directly. Therefore, bugs can propagate when an artifact invokes or extends another piece of code directly. For example, the bug LUCENE-3026 was caused by declaring a variable as short in *SegGraph.java*. This bug spread to its offspring *BiSegGraph.java* which reuses the same code, resulting in bug LUCENE-3049. If the correlated bug could be detected earlier, there would be no need to report the second bug again.

11:14 Do Bugs Propagate?

2) Bugfix: Bugfix can be another way to propagate bugs on dependent artifacts [43]. A bugfix for one piece of code may affect a dependent piece of code which has the same function and appears to invoke or co-change frequently with the ego [40]. For example, LUCENE-3505 was caused by *BooleanScorer2.freq()*. But the fix affected a dependent file *DisjunctionSumScorer.java*, leading to LUCENE-4401. In the discussion of LUCENE-4401, the developer mentioned such causality:

“I committed (also backported to 3.x branch, the bug does not affect any releases but would have affected unreleased 3.6.2 code, as it was caused by my previous bugfix: LUCENE-3505).”

The same causality happened in LUCENE-3631 and LUCENE-3855. As the developer mentioned in their discussion of LUCENE-3855:

“I (accidentally!!) caused this with LUCENE-3631, where we moved writable deletes from SegmentReader into IndexWriter.”

3) Human Factors: Another explanation of temporal bug correlations could be the developer interactions among artifacts [55, 18]. Bugs might propagate because the same careless developer makes the same mistake or clones buggy codes to elsewhere [48, 24]. For example, both LUCENE-2478 and LUCENE-3446 are bugs of missing *NullPointerException* checking. The LUCENE-2478 was found on May, 2010. The developer forgot to check null return of *Filter.getDocIdSet()* in the *CachingWrapperFilter.java*. Later, a similar mistake (LUCENE-3446) by the same developer was found on Sept, 2011. He did not check null return in the *BooleanFilter.java* again. We found both files had been changed together in revision 722174 in 2008. If we could predict the correlated bug and carefully test it in the *BooleanFilter.java*, the second bug would not happen.

A more sophisticated empirical study on the percentage of each mechanism could be added in future work.

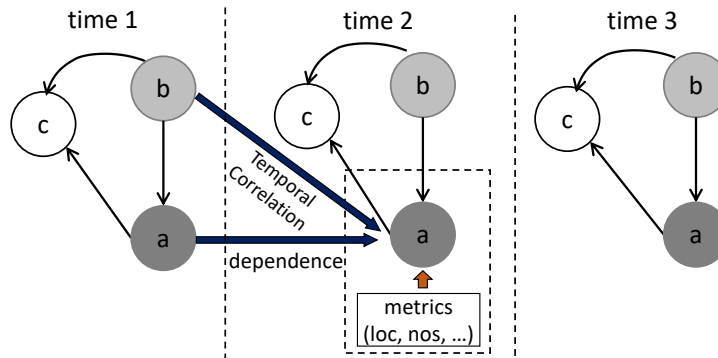
The temporal correlation among software bugs can be caused by code reuse, bugfix, and human factors.

6 Discussion

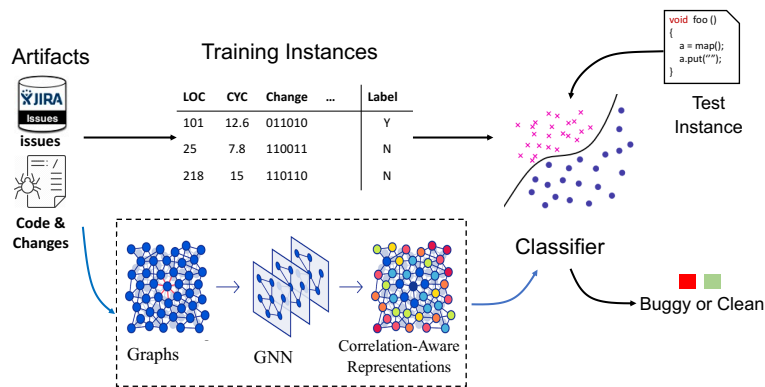
6.1 Application of Findings

In this section, we discuss potential applications of temporal bug correlations for practitioners. Our study and findings can inspire several research directions in the future.

- **Defect Prediction:** Traditional defect prediction techniques usually train a machine learning classifier with artifact metrics and make predictions by classifying new buggy instances. As Figure 4 summarizes, traditional defect factors include artifact metrics such as source code metrics (i.e., size, complexity), churn, authors [15, 32, 36]. These factors are related to individual artifacts and are time independent. Besides these traditional factors, researchers also found that bugs can be time-dependent. In other words, defects may linger in components that were recently defective [21]. We refer to the latter category of factor as “dependence”. In this paper, we have identified the third factor – temporal correlation. This means that neighbor bugs at previous time step may influence ego’s bug status. The temporal bug correlation and hybrid graph suggest a new factor to



■ **Figure 4** An illustration of factors to software defects. The three graphs from time 1 to time 3 represent three hybrid graphs in consecutive stages. Each circle stands for an artifact. The bold arrows represent different defect factors.



■ **Figure 5** An example of the propagation based defect prediction.

software defect prediction. We can enhance existing defect prediction model by combining temporal correlation with traditional defect metrics (i.e., LOC, Complexity). Besides simply adding temporal correlation as a new metric, we can apply the graph neural network (GNN) [31] to the proposed hybrid graph. GNN is a popular deep learning model for learning graph representations. As our hybrid graph incorporates significant information about bug correlations, it can be utilized to help the learning of defect prediction models. For example, a defect prediction model (Figure 5) could be developed, which runs the graph neural network on the hybrid graph to learn a correlation-aware representation for each artifact. The learnt representation, together with traditional metrics, are taken as input to the classifier for the prediction.

- **Linking Correlated Bug Reports:** Another possible application is to link correlated bug reports. Identifying related bug reports have been a key issue in software maintenance [20, 49, 42, 42]. Researchers have developed a variety of approaches to retrieve semantically related bug reports [49, 42]. With the property of temporal bug correlation, developers can link several inter-related bug reports based on both temporal and semantic correlation of bugs for efficient and effective fault diagnosis. For example, a series of bugs could be reported due to temporal correlations, where fixing one can help resolve others. Moreover, the links may represent other types of relevance such as re-occurrence or co-occurrence

11:16 Do Bugs Propagate?

- between two bugs. Recently, researchers have developed methods to predict semantically linkable Stack Overflow posts [57] and identify linked incident reports for online service systems [11], which have all confirmed the usefulness of linked knowledge.
- **Bug Localization.** Bug localization is one of the core tasks for software maintenance. Existing bug localization approaches often consider individual bug reports and identify buggy modules based on text similarities [33]. Some work also consider the historical similar bug reports [60], co-change histories [52], and the number of previous bugs [16]. The results of this research can provide more information, especially the temporal correlation information, to the existing bug localization work. Such information can help track the causal paths of bug and pinpoint their root cause, thus facilitating bug localization and triage.
 - **Architecture Refactoring.** The proposed hybrid graph provides a new perspective on software design and quality management. Developers can avoid temporal bug correlation in architectural design in order to improve the quality and reliability of software. For example, they can estimate the temporal correlations and then construct the hybrid correlation graph using our LLR-based analysis tool. Then, they can adjust design paradigms (e.g., UML graphs) by removing unnecessary artifact connections that have edges in the hybrid graph.

6.2 Threats to Validity

As a proof of concept, all projects investigated in our experiments are developed as JAVA open source projects. Although Java is one of the most popular programming languages, it might not be representative of commercial projects and projects written in other languages. However, our study is not limited to a certain language as it is conducted on software graphs which can be extracted from most languages. Investigating temporal bug correlations in other languages remains our future work.

Another threat lies in the selection of examination points. We selected examination points according to the selected releases. In fact, in some data sets, the intervals between releases vary a lot, leading to graphs that are either too dense or too sparse. We mitigated the threat by removing releases that are too close to others. Yet the intervals are still not equal. The selection of releases might affect our experimental results, however although the examination points are not selected with equal intervals, our results are still valid. This is because examination intervals for different subjects vary a lot but most of the results show the same trends. This means that altering the examination points does not change the results significantly.

7 Related Work

7.1 Bug Distribution and Dependence

Besides our work, there have been other studies that investigate bug distributions and find similar phenomena of this sort. For instance, Zimmerman et al. [61] analyzed the relationships between failures and dependent neighborhoods on dependence graphs. They found that depending on a component that has failures does not have an effect on failures of the dependent component in VISTA. Andersson [5] investigated how faults in large software systems are distributed over modules and discovered that the distribution of faults over modules follows the Pareto principle [28]. Zhang [59] found a Weibull distribution of bugs across packages in Eclipse system. Murgia et al. [40] represented Java software as artifact tie graphs and analyze the relationship between graph properties, statistic of software metrics,

and the distribution of bugs in such graphs. They found that the distribution of bugs across compliant units exhibits a power-law behavior in their tails, suggesting the spread of bugs in the software system. Mondal et al. [39] investigated bug propagation through code cloning. They define clone evolution patterns that reasonably indicate bug propagation through code cloning and found that up to 33% of the clone fragments that experience bug-fix changes can contain propagated bugs. Ai et al. [3] proposes a new software network model and analyzes the relationship between nodes or defects distribution and software network parameters, as well as high-risk module excavation through a defect density analysis.

Different from these studies, we conduct a deeper investigation on the dynamic view of software bugs across a wider range of software ties. In particular, we investigate time-lag bug correlations between software ties, design new ties (graph) exhibiting more severe bug correlation, and discuss the casual mechanisms behind this phenomenon.

7.2 Software Graphs

Besides our work, graph models have been widely used in empirical software engineering [7, 26, 58] especially for defect prediction [62, 63]. A typical use of graph models is in the analysis of developers' social networks [25, 55, 35]. Meneely et al. [37] leverage graphs to model collaborations among developers. They examine the graph metrics in a developers' collaboration network to predict failures. Pinzger et al. [45] modeled developer-modules as graphs. They found correlations between centrality measures of developer-module networks and failure-prone modules, and used such correlations to predict failures.

Dependency in graphs is also an active research topic. Zimmermann [62, 63] focused on the software dependency graph which considers interactions between software artifacts. They introduced both *ego networks* and *global networks*, using graphical metrics to enhance the performance of defect prediction. While most graph-based analysis studied social factors and program dependency information separately, Bird et al. [8] found that task assignment and dependency structure interact to influence the quality of the resulting software. Their prediction method combining social networks with dependency structures was able to predict failure proneness with better accuracy.

In addition to defect prediction, graphs are also widely used in other domains in Software Engineering such as the research of software evolution. To better observe the evolution of very large software systems, Pinzger et al. [44] proposed a visualization approach that can provide condensed graphical views on source codes and release history data for multiple releases. Bhattacharya [7] analyze the evolution of software using social network and graph metrics in several prediction tasks. Both these works made use of the dependence between software artifacts and leveraged the dependence as one of the metrics for defect prediction.

Despite the successful application of graph models in software engineering, these approaches often use macro-level graph metrics (e.g., degrees, centralities) directly without analyzing the defect correlations between graph nodes. Our study differs from these works by providing a deeper insight into micro-level bug correlation and propagation.

8 Conclusion

In this paper we applied correlation analysis from the social science literature to identify and measure the propagation of software bugs as a source of temporal correlations between software artifacts. We first investigated temporal bug correlation on known software ties such as calls, inheritance and co-changes. We performed longitudinal logistic regressions on the time-lag correlations between neighboring bugs on different artifact ties. Our empirical study with data from four open source projects showed that software bugs can be temporally

correlated among some known software ties. Based on our findings, we synthesize a hybrid graph (tie) which exhibits a higher degree of bug correlation.

All these findings shed light on new insights to software maintenance. In addition to metrics of individual artifacts, researchers can consider the factor of temporal bug correlation when designing the bug localization and prediction models.

In the future, we will investigate temporal bug correlations on more graphs. We will also apply the temporal bug correlation properties to improve related software engineering tasks such as defect prediction, correlated bug report linking, and bug localization. In particular, we will use the graph neural network (GNN) [31] on the proposed hybrid graph to learn the representation of bug correlations.

Our tool and experimental data described in this paper are available at <http://github.com/bugcorrelation/bugcorrelation>.

References

- 1 Understand, <http://www.scitools.com/>. URL: <http://www.scitools.com/>.
- 2 Wala project. <http://wala.sourceforge.net/>. URL: <http://wala.sourceforge.net/>.
- 3 Jun Ai, Wenzhu Su, Shaoxiong Zhang, and Yiwen Yang. A software network model for software structure and faults distribution analysis. *IEEE Transactions on Reliability*, 68(3):844–858, 2019.
- 4 Aris Anagnostopoulos, Ravi Kumar, and Mohammad Mahdian. Influence and correlation in social networks. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, pages 7–15, New York, NY, USA, 2008. ACM. doi:10.1145/1401890.1401897.
- 5 Carina Andersson and Per Runeson. A replicated quantitative analysis of fault distributions in complex software systems. *Software Engineering, IEEE Transactions on*, 33(5):273–286, 2007.
- 6 Sinan Aral, Lev Muchnik, and Arun Sundararajan. Distinguishing influence-based contagion from homophily-driven diffusion in dynamic networks. *Proceedings of the National Academy of Sciences*, 106(51):21544–21549, 2009. doi:10.1073/pnas.0908800106.
- 7 Pamela Bhattacharya, Marios Iliofotou, Iulian Neamtiu, and Michalis Faloutsos. Graph-based analysis and prediction for software evolution. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 419–429, Piscataway, NJ, USA, 2012. IEEE Press. URL: <http://dl.acm.org/citation.cfm?id=2337223.2337273>.
- 8 C. Bird, N. Nagappan, H. Gall, B. Murphy, and P. Devanbu. Putting it all together: Using socio-technical networks to predict failures. In *Software Reliability Engineering, 2009. ISSRE '09. 20th International Symposium on*, pages 109–119, November 2009. doi:10.1109/ISSRE.2009.17.
- 9 John T Cacioppo, James H Fowler, and Nicholas A Christakis. Alone in the crowd: the structure and spread of loneliness in a large social network. *Journal of personality and social psychology*, 97(6):977, 2009.
- 10 Peter J Carrington, John Scott, and Stanley Wasserman. *Models and methods in social network analysis*. Cambridge university press, 2005.
- 11 Yujun Chen, Xian Yang, Hang Dong, Xiaoting He, Hongyu Zhang, Qingwei Lin, Junjie Chen, Pu Zhao, Yu Kang, Feng Gao, Zhangwei Xu, and Dongmei Zhang. Identifying linked incidents in large-scale online service systems. In *Proceedings of the 2020 ESEC/FSE*. ACM, 2020.
- 12 Nicholas A Christakis and James H Fowler. The spread of obesity in a large social network over 32 years. *New England journal of medicine*, 357(4):370–379, 2007.
- 13 Nicholas A Christakis and James H Fowler. Social contagion theory: examining dynamic social networks and human behavior. *Statistics in medicine*, 32(4):556–577, 2013.
- 14 M. D'Ambros, M. Lanza, and R. Robbes. On the relationship between change coupling and software defects. In *Reverse Engineering, 2009. WCRE '09. 16th Working Conference on*, pages 135–144, October 2009. doi:10.1109/WCRE.2009.19.

- 15 Marco D'Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 31–41. IEEE, 2010.
- 16 S. Davies and M. Roper. Bug localisation through diverse sources of information. In *2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 126–131, 2013.
- 17 Harpal Dhama. Quantitative models of cohesion and coupling in software. *Journal of Systems and Software*, 29(1):65–74, 1995.
- 18 Dario Di Nucci, Fabio Palomba, Giuseppe De Rosa, Gabriele Bavota, Rocco Oliveto, and Andrea De Lucia. A developer centered bug prediction model. *IEEE Transactions on Software Engineering*, 44(1):5–24, 2017.
- 19 James H. Fowler, Nicholas A. Christakis, Steptoe, and Diez Roux. Dynamic spread of happiness in a large social network: Longitudinal analysis of the framingham heart study social network. *BMJ: British Medical Journal*, 338(7685):pp. 23–27, 2009. URL: <http://www.jstor.org/stable/20511686>.
- 20 Katerina Goseva-Popstojanova and Jacob Tyo. Identification of security related bug reports via text mining using supervised and unsupervised classification. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 344–355. IEEE, 2018.
- 21 Todd L Graves, Alan F Karr, James S Marron, and Harvey Siy. Predicting fault incidence using software change history. *Software Engineering, IEEE Transactions on*, 26(7):653–661, 2000.
- 22 Geoffrey Hecht, Omar Benomar, Romain Rouvoy, Naouel Moha, and Laurence Duchien. Tracking the software quality of android applications along their evolution (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 236–247. IEEE, 2015.
- 23 Kim Herzig, Sascha Just, and Andreas Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 392–401. IEEE Press, 2013.
- 24 Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 121–130. IEEE Press, 2013.
- 25 Qiaona Hong, Sunghun Kim, SC Cheung, and Christian Bird. Understanding a developer social network and its evolution. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 323–332. IEEE, 2011.
- 26 Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. Improving bug triage with bug tossing graphs. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 111–120. ACM, 2009.
- 27 Tian Jiang, Lin Tan, and Sunghun Kim. Personalized defect prediction. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 279–289, November 2013. doi:10.1109/ASE.2013.6693087.
- 28 JM Juran and FM Gryna. Juran's quality control handbook. NY: McGraw-Hill, 1988.
- 29 David Kempe, Jon Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '03*, pages 137–146, New York, NY, USA, 2003. ACM. doi:10.1145/956750.956769.
- 30 Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, pages 489–498. IEEE Computer Society, 2007.
- 31 Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

11:20 Do Bugs Propagate?

- 32 Akif Günes Koru, Dongsong Zhang, Khaled El Emam, and Hongfang Liu. An investigation into the functional form of the size-defect relationship for software modules. *Software Engineering, IEEE Transactions on*, 35(2):293–304, 2009.
- 33 An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 476–481. IEEE, 2015.
- 34 Kung-Yee Liang and Scott L Zeger. Longitudinal data analysis using generalized linear models. *Biometrika*, 73(1):13–22, 1986.
- 35 Wanwangying Ma, Lin Chen, Yibiao Yang, Yuming Zhou, and Baowen Xu. Empirical analysis of network measures for effort-aware fault-proneness prediction. *Information and Software Technology*, 69:50–70, 2016.
- 36 Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, pages 308–320, 1976.
- 37 Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. Predicting failures with developer networks and social network analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '08/FSE-16*, pages 13–23, New York, NY, USA, 2008. ACM. doi:10.1145/1453101.1453106.
- 38 Ran Mo, Yuanfang Cai, Rick Kazman, Lu Xiao, and Qiong Feng. Architecture anti-patterns: Automatically detectable violations of design principles. *IEEE Transactions on Software Engineering*, 2019.
- 39 Manishankar Mondal, Chanchal K Roy, and Kevin A Schneider. Bug propagation through code cloning: An empirical study. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 227–237. IEEE, 2017.
- 40 Alessandro Murgia, Giulio Concas, Michele Marchesi, Roberto Tonelli, and Ivana Turnu. An analysis of bug distribution in object oriented systems. *arXiv preprint arXiv:0905.3296*, 2009.
- 41 N. Nagappan and T. Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 364–373, September 2007. doi:10.1109/ESEM.2007.13.
- 42 Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N Nguyen, David Lo, and Chengnian Sun. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 70–79. IEEE, 2012.
- 43 Zhen Ni, Bin Li, Xiaobing Sun, Tianhao Chen, Ben Tang, and Xinchun Shi. Analyzing bug fix for automatic bug cause classification. *Journal of Systems and Software*, 163:110538, 2020.
- 44 Martin Pinzger, Harald Gall, Michael Fischer, and Michele Lanza. Visualizing multiple evolution metrics. In *Proceedings of the 2005 ACM symposium on Software visualization*, pages 67–75. ACM, 2005.
- 45 Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. Can developer-module networks predict failures? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '08/FSE-16*, pages 2–12, New York, NY, USA, 2008. ACM. doi:10.1145/1453101.1453105.
- 46 Sarah J. Ratcliffe and Justine Shults. GEEQBOX: A matlab toolbox for generalized estimating equations and quasi-least squares. *Journal of Statistical Software*, 25(14):1–14, May 2008. URL: <http://www.jstatsoft.org/v25/i14>.
- 47 H. Sajjani, V. Saini, and C. V. Lopes. A comparative study of bug patterns in java cloned and non-cloned code. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 21–30, 2014.
- 48 Gehan MK Selim, Liliane Barbour, Weiyi Shang, Bram Adams, Ahmed E Hassan, and Ying Zou. Studying the impact of clones on software defects. In *2010 17th Working Conference on Reverse Engineering*, pages 13–21. IEEE, 2010.

- 49 Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. Towards more accurate retrieval of duplicate bug reports. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 253–262. IEEE, 2011.
- 50 Chenhao Tan, Jie Tang, Jimeng Sun, Quan Lin, and Fengjiao Wang. Social action tracking via noise tolerant time-varying factor graphs. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '10*, pages 1049–1058, New York, NY, USA, 2010. ACM. doi:10.1145/1835804.1835936.
- 51 Jie Tang, Jimeng Sun, Chi Wang, and Zi Yang. Social influence analysis in large-scale networks. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '09*, pages 807–816, New York, NY, USA, 2009. ACM. doi:10.1145/1557019.1557108.
- 52 C. Tantithamthavorn, A. Ihara, and K. Matsumoto. Using co-change histories to improve bug localization performance. In *2013 14th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, pages 543–548, 2013.
- 53 Justin G Trogon, James Nonnemaker, and Joanne Pais. Peer effects in adolescent overweight. *Journal of health economics*, 27(5):1388–1399, 2008.
- 54 Ye Wang, Na Meng, and Hao Zhong. An empirical study of multi-entity changes in real bug fixes. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 287–298. IEEE, 2018.
- 55 Timo Wolf, Adrian Schroter, Daniela Damian, and Thanh Nguyen. Predicting build failures using social network analysis on developer communication. In *Proceedings of the 31st International Conference on Software Engineering*, pages 1–11. IEEE Computer Society, 2009.
- 56 Lu Xiao, Yuanfang Cai, and Rick Kazman. Design rule spaces: A new form of architecture insight. In *Proceedings of the 36th International Conference on Software Engineering*, pages 967–977, 2014.
- 57 Bowen Xu, Deheng Ye, Zhenchang Xing, Xin Xia, Guibin Chen, and Shanping Li. Predicting semantically linkable knowledge in developer online forums via convolutional neural network. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 51–62. IEEE, 2016.
- 58 Marcelo Serrano Zanetti, Ingo Scholtes, Claudio Juan Tessone, and Frank Schweitzer. Categorizing bugs with social networks: A case study on four open source software communities. In *Proceedings of the 2013 International Conference on Software Engineering, ICSE '13*, pages 1032–1041, Piscataway, NJ, USA, 2013. IEEE Press. URL: <http://dl.acm.org/citation.cfm?id=2486788.2486930>.
- 59 Hongyu Zhang. On the distribution of software faults. *IEEE Transactions on Software Engineering*, pages 301–302, 2007.
- 60 J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 14–24, 2012.
- 61 Tom Zimmerman, Nachiappan Nagappan, Kim Herzig, Rahul Premraj, and Laurie Williams. An empirical study on the relation between dependency neighborhoods and failures. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 347–356. IEEE, 2011.
- 62 Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th International Conference on Software Engineering, ICSE '08*, pages 531–540, New York, NY, USA, 2008. ACM. doi:10.1145/1368088.1368161.
- 63 Thomas Zimmermann and Nachiappan Nagappan. Predicting defects with program dependencies. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, ESEM '09*, pages 435–438, Washington, DC, USA, 2009. IEEE Computer Society. doi:10.1109/ESEM.2009.5316024.