Is Late Propagation a Harmful Code Clone Evolutionary Pattern? An Empirical Study

Osama Ehsan, Lillane Barbour, Foutse Khomh, and Ying Zou

Abstract Two similar code segments, or clones, form a clone pair within a software system. The changes to the clones over time create a clone evolution history. Late propagation is a specific pattern of clone evolution. In late propagation, one clone in the clone pair is modified, causing the clone pair to become inconsistent. The code segments are then re-synchronized in a later revision. Existing work has established late propagation as a clone evolution pattern, and suggested that the pattern is related to a high number of faults. In this chapter, we replicate and extend the work by Barbour et al. (2011 27th IEEE International Conference on Software Maintenance (ICSM). IEEE (2011) [1]) by examining the characteristics of late propagation in 10 long-lived open-source software systems using the iClones clone detection tool. We identify eight types of late propagation and investigate their fault-proneness. Our results confirm that late propagation is the more harmful clone evolution pattern and that some specific cases of late propagations are more harmful than others. We trained machine learning models using 18 clone evolution related features to predict the evolution of late propagation and achieved high precision within the range of 0.91–0.94 and AUC within the range of 0.87–0.91.

O. Ehsan (✉) · L. Barbour · Y. Zou
Queen's University, Kingston, Canada
e-mail: osama.ehsan@queensu.ca

L. Barbour
e-mail: l.barbour@queensu.ca

Y. Zou
e-mail: ying.zou@queensu.ca

F. Khomh
Polytechnique Montréal, Quebec City, Canada
e-mail: foutse.khomh@polymtl.ca

K. Inoue and C. K. Roy (eds.), *Code Clone Analysis*,
https://doi.org/10.1007/978-981-16-1927-4_11

1 Introduction

A code segment is labeled as a code clone if it is identical or highly similar to another code segment. Similar code segments form a clone pair. Clone pairs can be introduced into systems deliberately (e.g., "copy-and-paste" actions) or inadvertently by a developer during development and maintenance activities. Like all code segments, code clones are not immune to change. Large software systems undergo thousands of revisions over their lifecycles. Each revision can involve modifications to code clones. As the clones in a clone pair are modified, a change evolution history, known as a clone genealogy [2], is generated.

In a previous study on clone genealogies, Kim et al. [2] define two types of evolutionary changes that can affect a clone pair: a consistent change or an inconsistent change. During a consistent change, both clones in a clone pair are modified in parallel, preserving the clone pair. In an inconsistent change, one or both of the clones evolve independently, destroying the clone pair relationship. Inconsistent changes can occur deliberately, such as when code is copied and pasted and then subsequently modified to fit the new context. For example, if a driver is required for a new printer model, a developer could copy the driver code from an older printer model and then modify it. Inconsistent changes can also occur accidentally. A developer may be unaware of a clone pair and cause an inconsistency by changing only one half of the clone pair. This inconsistency could cause a software fault. If a fault is found in one clone and fixed, but not propagated to the other clone in the clone pair, the fault remains in the system. For example, a fault might be found in the old printer driver code and fixed, but the fix is not propagated to the new printer driver. For these reasons, a previous study [2] has argued that accidental inconsistent changes make code clones more prone to faults.

Late propagation occurs when a clone pair undergoes one or more inconsistent changes followed by a re-synchronizing change [3]. The re-synchronization of the code clones indicates that the gap in consistency is accidental. Since accidental inconsistencies are considered risky [4], the presence of late propagation in clone genealogies can be an indicator of risky, fault-prone code.

Many studies have been performed on the evolution of clones. A few (e.g., [3, 4]) have studied late propagation and indicated that late propagation genealogies are more fault-prone than other clone genealogies. Thummalapenta et al. began the initial work in examining the characteristics of late propagation. The authors measured the delay between an inconsistent change and a re-synchronizing change and related the delay to software faults. In our chapter, we examine more characteristics of late propagation to determine if only a subset of late propagation genealogies are at risk of faults. Developers are interested in identifying which clones are most at risk of faults. Our goal is to support developers in their allocation of limited code testing and review resources toward the most risky late propagation genealogies. To achieve this goal, we first study the prevalence and fault-proneness of late propagation genealogies, and secondly we train multiple machine learning models to predict whether a clone pair

would have late propagation. Early diagnosis of late propagation can help developers in addressing the clones with late propagation fast before they become buggy.

In this chapter, we replicate and extend the analysis of late propagation performed by Barbour et al. [1]. We study the characteristics of late propagation genealogies and estimate the likelihood of faults. We used 10 open-source projects from GitHub instead of only two projects as in the original study. We also include an additional research question aimed at predicting occurrences of late propagation genealogies.

2 Experimental Setup

The *goal* of our study is to investigate the fault-proneness of clone pairs that undergo late propagation. The *quality focus* is to lower the maintenance effort and cost due to the presence of late propagated clone pairs in software systems. The *perspective* is that of researchers interested in studying the effects of late propagation on clone pairs. The results may also be of interest to developers who perform development or maintenance activities. The results will provide insight in deciding which code segments are most at risk for faults and in prioritizing the code for testing.

The *context* of this study consists of the change history of open-source software projects, which have different sizes and belong to different domains. This section describes the setup used to perform our study which aims to address the following four research questions:

- RQ1: Are there different types of late propagation?,
- RQ2: Are some types of late propagation more fault-prone than others?,
- RQ3: Which type of late propagation experiences the highest proportion of faults?, and
- RQ4: Can we predict whether a clone pair would experience late propagation?.

2.1 Project Selection

We use GHTorrent on the Google Cloud[1] to extract all projects that have more than 1,000 commits, 1,000 issues, and 1,000 pull requests. We use such a high number of commits, pull requests, and issues to ensure that we have enough history of clone genealogies. We limit our study to Java projects. Our selection criteria provide us with 66 Java projects. Then, we discard the projects that are younger than 5 years (created after June 2015). If a project has more source lines of code (SLOC), the probability of having code clones increases. A recent study suggests [5] to include projects with more than 100K source lines of code. We remove the projects with less than 100K

[1] https://ghtorrent.org/gcloud.html.

Table 1 Description of selected projects

Project name	# of commits	# of issues	SLOC	% of java files (%)	# of clone genealogies
Druid	10,496	1,657	1.2m	94.50	61,718
Netty	9,910	4,174	476.2k	98.60	6,576
Muikku	16,970	2,696	318.4k	50.4	23,836
Framework	18,969	1,788	867.9k	95.50	11,961
Checkstyle	9,454	2,198	457.4k	97.80	7,705
Gatk	4,173	2,736	2.2m	93.70	22,651
Realm	8,318	3,358	199.9k	83.80	13,540
Nd4j	7,021	1,238	467.0k	99.80	45,413
Rxjava	5,762	1,950	474.9k	99.90	8,866
K	15,997	1,134	243.3k	83.50	6,026

SLOC by using the GitHub project SLOC calculator extension.[2] Furthermore, we remove the forked projects and the projects which have less than 70% of Java files. The percentage of Java files is calculated using GitHub's language information of each project. Finally, after applying all the selection criteria, we retain the top 10 projects used in this study. Table 1 provides the description of the selected projects.

2.2 Building Clone Genealogies

The selected projects are all Git-based projects. Git provides multiple functions to extract the history of the projects. The history includes the renamed files, changed files, and changes made to each file using the `blame` function. We perform the following steps on each of the projects in our dataset. After downloading the repositories, we use the following command to extract the identifier, committer email, commit date, and the message of each commit:

git log – pretty=format:"%h,%ae, %ai, %s"

2.2.1 Detecting Code Clones

We use the latest version of the iCLONES clone detection [6] to identify the clones from the projects. We select iCLONES because it is recommended by Svajlenko et al. [7] who evaluate the performance of 11 different clone detection tools. iCLONES uses a hybrid approach to detect clones. We use the settings used by Svajlenko et al. [7] as the recommended settings are reported to achieve higher precision and recall

[2] https://github.com/artem-solovev/gloc.

values. We use the `git checkout` command to extract a snapshot of a project at a specific commit. We sort all commits chronologically and run the clone detection on each commit.

2.2.2 Extracting Clone Genealogies

Code clones may experience changes during the development and maintenance phases of the project. Such changes can be consistent or inconsistent based on a relative similarity score. Inconsistent clones can be later synchronized to become consistent. The set of states and the history of changes to any clone pairs are known as clone pair genealogy. We identify clone genealogies of all the clones in the studied projects. Our approach for generating clone genealogies is similar to the approaches used in other studies [8, 9]. Both Göde and Krinke track clones over time by acquiring a list of changes from the source code repositories of the subject systems.

The iCLONES tool produces a list of clones that exist in a project at any specific commit. We link the clone pairs between each commit to create a set of genealogies. A change to a clone can affect its size while a change to a file containing the clone can shift the position of the clone (i.e., changes its line numbers). To address this issue, we use the `git diff` command to detect all the changes to a specific file. We track the clone positional changes affected by the changes to the non-clone part of the file. We include only the changes to the clone contents rather than the clone line number since a shift in the line numbers does not change the state of the clone.

We build a clone genealogy for each clone pair detected by the iCLONES tool. We start by extracting the commit sequence of each project under study. We use the commit sequence to identify the modifications in the clone pairs of each commit. If a commit C2 changes a file that contains code in the clone pair, we use the `diff` command to compare the changes to a previous commit C1. If a clone snippet is changed in C2, we update the start and end line numbers of the clone from C2. To generate the mapping and to check the modifications, we used a third-party Python patching parser called `whatthepatch` [10]. If the start or the end of the clone snippet is deleted, we move the clone line numbers accordingly to address the deleted lines. Krinke [9] made several assumptions when updating line numbers of clones between revisions. We use the same assumptions in our study:

1. If a change occurs before the start of the clone, or after the end of the clone, the clone is not modified.
2. If an addition occurs starting at the first line number of a clone, the clone shifts within the method but is not modified.
3. If a deletion occurs anywhere within the clone boundaries, the clone is modified and its size shrinks.
4. If a deletion followed by an addition overlaps the clone boundaries, we assume that the clone size shrinks because of the deletion, and the new lines do not makeup part of the clone.

In the last assumption, it is possible that there exists a clone containing both our updated reference clone and the newly added lines. We use the strictest assumption that the new lines are not included. When determining consistent and inconsistent changes, we look for clones in the clone list that *contain* our updated reference clone. Therefore, this scenario would still be considered a consistent change. In addition, we also track changes to the names of the clone files.

2.3 Classification of Genealogies

In the current state of the art, late propagation is defined as a clone pair that experiences one or more inconsistent changes followed by a re-synchronizing change [4]. For example, consider two clones that call a method. A developer modifies the call parameters of the method and updates one of the clones to reflect the change. This causes the clone pair to become inconsistent. Using all combinations of the inconsistent phases described by Barbour et al. [1], we identify eight possible types of late propagation (LP) genealogies. The detail of the eight types of late propagation are described in [1]. The eight types are organized in three groups based on the occurrence or not of a change propagation: (1) propagation always occurs (three types named LP1, LP2, and LP3), (2) propagation may or may not occur (four types named LP4, LP5, LP6, and LP7), and (3) propagation never occurs (one type named LP8). In this study, we examine if the cases that always involve propagation (i.e., LP1, LP2, and LP3) or never involve propagation (i.e., LP8) are more prone to faults than the other types of late propagation. We made a slight modification in the definition of LP7 to include cases where during divergence either A or B is changed, instead of considering only instances in which both A and B are changed during divergence.

2.4 Detecting Faulty Clones

We use the SZZ algorithm [11] to identify the changes that introduced faults. First, we use the Fischer et al. [12] heuristic to identify fault-fixing commits using a regular expression. The regular expression identifies the bug-ID in the commit messages. If a bug-ID appears in the commit message, we map the commit to the bug as a bug-fixing commit. Second, we mine the issue reports of each project from GitHub. For the issues that are closed, we identify if there are any pull requests associated with such issues. If there is a pull request associated with an issue, we identify all the commits included in the pull request and map the commits to the issue as a bug-fixing commit. Once we have a list of all bug-fixing commits, we use the following command to identify all the modified files in each commit.

git log [commit-id]-n 1—name-status

We consider only changes to Java files in a commit. A commit is a set of changes to the file(s) in the software repository. For all changes to a specific file of a bug-fixing commit, we use the `git blame` command to identify all the commits when the same snippet was changed. We consider such commits as the "candidate faulty changes." We exclude the changes that are blank lines or comments.

Finally, we filter the commits that are submitted before the creation date of the bugs. We then check whether the commits identified as bug-inducing commits include clone pairs. If a clone snippet is included in the bug-inducing commits, we label the clone change as "buggy."

3 Case Study Results

This section reports and discusses the results of our study.

3.1 RQ1: Are There Different Types of Late Propagation?

Motivation. This question is preliminary to questions RQ2 and RQ3. It provides quantitative data on the percentages with which different types of late propagation occur in our studied systems.

Approach. We address this question by classifying all instances of late propagation as described in Sect. 2.3. For each type of late propagation, we report the number of occurrences in the systems. Table 2 lists each of the categories and the proportion of occurrences in our dataset, both as a numerical value and a percentage of the overall number of late propagation instances for the systems.

Results. As summarized in Table 2, four types of late propagation are dominant across all systems when using the iClones clone detection tool (i.e., LP1, LP3, LP7, and LP8). The four dominant types represent the three late propagation categories. Only LP3 (instead of LP6) is more dominant as compared to the results of Barbour et al. [1]. As shown in Table 2, LP7 occurs in an average of 40.5% of instances of late propagation, so it is the most common form of late propagation across all systems. However, LP7 is also the least understood of the types of late propagation. Since both clones in LP7 clone pairs can be modified during all three steps of late propagation (i.e., experiencing a diverging change, a change during the period of divergence, a re-synchronizing change), it is unclear in which direction changes are propagated during the evolution of the clone pair. A few types of late propagation (i.e., LP2, LP4, and LP5) contribute minutely to the number of late propagation genealogies. Other than the one project (Muikku), all the other projects include almost all types of late propagation. Our further investigation shows that only 1% (297 out of 23,836)

Table 2 Summary of late propagation types for the studied open-source projects

Propagation category	LP type	Projects										Total	%
		Druid	Netty	Muikku	Framework	Checkstyle	Gatk	Realm	Nd4j	RxJava	K		
Propagation always occurs	LP1	102	15	78	46	3	102	39	74	10	21	490	13
	LP2	23	–	4	5	5	3	3	5	–	4	52	1.5
	LP3	195	22	–	24	14	11	57	67	3	58	451	12
Propagation may or may not occur	LP4	18	5	10	7	6	1	4	15	1	6	73	2
	LP5	49	3	–	3	10	3	19	24	3	6	120	3
	LP6	207	12	–	12	10	7	28	76	6	19	377	10
	LP7	714	62	–	29	81	21	143	297	78	102	1,527	40.5
Propagation never occurs	LP8	102	18	210	29	3	195	28	50	2	53	690	18
Total LPs		1,410	137	302	155	132	343	321	608	103	269	3,780	100

of the clone genealogies experience late propagation which is the lowest among all the projects and this might be the reason for the absence of half of LP types.

Overall, we conclude that there is representation from multiple types of late propagation and across all categories of late propagation. In the next two research questions, we examine the types in more detail to determine if some types are more risky than others.

Summary of RQ1

Late propagation types LP1, LP3, LP7, and LP8 are the most commonly occurring type of late propagation in the 10 studied open-source projects from GitHub. The results are consistent with the previous study except that LP3 is more frequent instead of LP6. Most of the projects include all types of late propagations.

3.2 *RQ2: Are Some Types of Late Propagation More Fault-Prone than Others?*

Motivation. Previous researchers have determined that late propagation is more prone to faults than other clone genealogies [3]. Using the classification of late propagation clone genealogies proposed by Barbour et al. [1], we evaluate late propagation in greater depth and examine if the risk of faults remains consistent across all types of late propagation.

Approach. We compute the number of fault-containing and fault-free genealogies in each late propagation category. We compute the same values for non-late propagation clone genealogies that experience at least one change. For the remainder of this chapter, we use the abbreviation "Non-LP" for clone pairs that experience at least one change but are not involved in any type of late propagation. We test the following null hypothesis[3] H_{02}: *Each type of late propagation genealogy has the same proportion of clone pairs that experience a fault fix*.

We use the Chi-square test [13] and compute the *odds ratio* (OR) [13]. The Chi-square test is a statistical test used to determine if there are non-random associations between two categorical variables. The odds ratio indicates the likelihood of an event to occur. It is defined as the ratio of the odds p of an event (i.e., fault-fixing change) occurring in one sample (i.e., experimental group), to the odds q of the event occurring in the other sample (i.e., control group): $\text{OR} = \frac{p/(1-p)}{q/(1-q)}$. An $OR = 1$ indicates that the event is equally likely in both samples; an $OR > 1$ shows that the event is more likely in the experimental group while an $OR < 1$ indicates that it is more likely in the control group. Specifically, we compute two sets of odds ratios. First,

[3] There is no H_{01} because RQ1 is exploratory.

Table 3 Contingency table, Chi-square tests results for clone genealogies with and without late propagation. The table shows the values for all the combinations of late propagations and faults

LP-faults	LP-no faults	No LP-faults	No LP-no faults	p-value	OR
1,851	1,929	42,526	48,928	<0.05	1.8

we select the clone pairs that underwent a late propagation as an experimental group. Second, we form one experimental group for each type LP_i of late propagation and re-compute the odds ratios. In both cases, we select the non-LP genealogies as the control group.

Results. Previous researchers [4] have studied the relationship between late propagation and faults. In this research question, we first replicate the earlier studies and then extend the study to include the different categories of late propagation.

(a) Fault-proneness of late propagation. Table 3 summarizes the results of the tests described above for instances of late propagation compared to non-late propagation (LP) genealogies. The first and second columns show the number of LP genealogies with and without faults. The third and fourth columns in the table list the number of non-LP genealogies that experience fault fixes and the number that is free of fault fixes. The last column of the table lists the odds ratio test results for each system. All of our results pass the Chi-square test with a p-value less than 0.05 and are therefore significant. Where there are few data points, we use Fisher's exact test to confirm the results from the Chi-Square test. Fisher's exact test is more accurate than the Chi-square test when sample sizes are small [13]. In this study, the Fisher test provides the same information as the Chi-square test, so we do not present the Fisher test results in the tables. Table 4 shows the percentage of fault-prone late propagation in each of the studied projects. In all the significant cases, the odds ratio is greater than 1, indicating that late propagation genealogies are more fault-prone than non-LP genealogies. Overall, our results agree with previous studies [4] that found that late propagation is more at risk of faults.

(b) Fault-proneness of late propagation types. We repeat the previous tests, dividing the instances of late propagation into their respective late propagation types. We compare each type of late propagation to genealogies with no late propagation. For each type of late propagation, Table 5 lists the number of instances that experience a fault fix, the number of instances with a no-fault fix, the result from the Chi-square test, and the odds ratio using the control group composed of non-LP genealogies.

An examination of the significant cases in Tables 5 reveals that the odds ratios are greater than 1, so each type of late propagation is more fault-prone than non-LP genealogies. There are two exceptions to this observation, LP2 and LP3 in Table 5. All exceptions belong to the "propagation always occurs" category. Thus, in general, these late propagation types are not more fault-prone than non-LP genealogies. Our observation is consistent with the previous findings by Barbour et al. [1].

Table 4 Contingency table, Chi-square tests results for clone genealogies with and without late propagation

Projects	LP-faults	LP-no faults	% of faulty LPs (%)
Druid	970	440	67
Netty	1	136	0.5
Muikku	145	157	48
Framework	3	152	2
Checkstyle	78	54	60
Gatk	134	209	39
Realm	135	186	42
Nd4j	283	325	47
Rxjava	0	103	0
K	102	167	38

Table 5 Contingency table with the Chi-square test for different late propagation types

Propagation category	LP type	Faults	No faults	p-value	OR
	No LP	42,526	48,928	<**0.01**	1
Propagation always occurs	LP1	244	246	< **0.01**	3.953
	LP2	20	32	<**0.01**	0.672
	LP3	224	227	<**0.01**	0.922
Propagation may or may not occur	LP4	23	50	< **0.01**	2.256
	LP5	68	52	<**0.01**	1.765
	LP6	216	161	<**0.01**	6.179
	LP7	803	724	<**0.01**	1.277
Propagation never occurs	LP8	253	437	<**0.01**	3.2

We conclude that there are many types that make up a small proportion of LP instances and have a very high odds ratio. Thus, when one of these LP types occurs, the risk of fault introduction is high. For example, LP6 has a high odds ratio (e.g., 6.17 in Table 5) but accounts for less than 5% of all late propagation instances in Table 2.

The two most common late propagation types in the previous research question, LP7 and LP8, in general, have low odds ratios in Table 5. This indicates that although they occur frequently, they are less fault-prone than other less common late propagation types (e.g., LP6). The result is consistent with the previous findings by Barbour et al. [1]. Overall, each type of late propagation has a different level of fault-proneness. Thus, we reject H_{02} in general.

Summary of RQ2

The most commonly occurring late propagation types (i.e., LP7 and LP8) are less fault-prone than the less commonly occurring late propagation (i.e., LP6). The result is consistent with the previous study and shows that each propagation type is different from others.

3.3 RQ3: Which Type of Late Propagation Experiences the Highest Proportion of Faults?

Motivation. In the previous research question (i.e., RQ2), we identify the fault-proneness of late propagation types as compared to the no-LP clone pairs. The results show that fault-proneness is not related to the frequency of LP type. In this research question, we want to identify which type of late propagation experiences the highest proportion of faults. In other words, we examine if, when faults occur, do they occur in large numbers?

Approach. We test the following null hypothesis H_{03}: *Different types of late propagation have the same proportion of clone pairs that experience a fault fix*. For each type of late propagation, we calculate the sum of all faults experienced by instances of that type of late propagation. We use the non-parametric Kruskal–Wallis test to investigate if the number of faults for the different types of late propagation is identical.

Results. Table 6 presents the distribution of faults for different types of late propagation. The "Total" row represents the total numbers of faults over all late propagation genealogies. To validate the results, we perform the non-parametric Kruskal–Wallis test which compares the distribution of faults between groups of different types of late propagation. The results of the Kruskal–Wallis test is statistically significant with a p-value of 2.89^{-15}. Hence, there is a statistically significant difference between the distribution of faults across all types of late propagations.

Examining the results in Table 6 for the significant cases, we see that, in general, LP7 and LP8 contribute to a large proportion of the faults. In the previous question, LP7 and LP8 have lower odds ratios. Although they are less prone to faults, when they do experience faults, the faults are likely to occur in large numbers. The change causing the inconsistency may lead to faults in the system, which may explain why the change is reverted instead of being propagated to the other clone in the clone pair. Overall, we can conclude that types LP7 and LP8 are the most dangerous. The level of fault-proneness of the other types is system-dependant. The proportion of faults for each type of late propagation is, therefore, very different. Thus, we reject H_{03}. This result is consistent with the findings of Barbour et al. [1].

Table 6 Proportion of faults for each type of late propagation

Propagation category	LP type	# of faults	% of faults (%)
Propagation always occurs	LP1	244	13.2
	LP2	20	1.1
	LP3	224	12
Propagation may or may not occur	LP4	23	1.2
	LP5	68	3.7
	LP6	216	11.7
	LP7	803	43.3
Propagation never occurs	LP8	253	13.8
	TOTAL	1851	100.00

Summary of RQ3

In terms of the proportion of faults, LP7 and LP8 are more risky and should be monitored carefully and/or refactored if possible. The risk for the other types of late propagation is system-dependant.

3.4 *RQ4: Can We Predict Whether a Clone Pair Would Experience Late Propagation?*

Motivation. In this research question, we use machine learning algorithms to train models that can help developers predict which clone pair will experience late propagation and have faults in the future. Using these predictions, developers would be able to refactor risky clone pair early on and/or keep them in check before the clone pair becomes inconsistent or a fault is introduced. This information about risky clone pairs will help developers in making better use of their time and resources.

Approach. For each instance in the clone pair genealogy, we calculate multiple features that may help with training the models for predicting whether a clone pair would experience late propagation or not. The features are used in a prior study by Barbour et al. [5]. Table 7 presents the description of our collected features.

We train models for two different behaviors; (1) presence of late propagation (M_{LP}) and (2) fault-prone late propagation (M_{BUG}). For every change experienced by a clone pair, we calculate 18 features as described in Table 7. We also examined the fault-proneness of the clone pairs, as described in Sect. 2.

Table 7 Description of clone genealogies features from [5] used to build the models

Metric	Description
Product metrics	
CLOC	The number of cloned lines of code
CPathDepth	The number of common folders within the project directory structure
CCurSt	The current state of the clone pair (consistent or inconsistent)
CommiterExp	The experience of committer (i.e., the number of previous commits submitted before a specific commit.)
Process metrics	
EFltDens	The number of fault fix modifications to the clone pair since it was created divided by the total number of commits that modified the clone pair
TChurn	The sum of added and the changed lines of code in the history of a clone
TPC	The total number of changes in the history of a clone
NumOfBursts	The number of change bursts on a clone. A change burst is a sequence of consecutive changes with a maximum distance of one day between the changes
SLBurst	The number of consecutive changes in the last change burst on a clone
CFltRate	The number of fault-prone modifications to the clone pair divided by the total number of commits that modified the clone pair
Genealogy metrics	
EConChg	The number of consistent changes experienced by the clone pair
EIncChg	The number of inconsistent changes experienced by the clone pair
EConStChg	The number of consistent change of state within the clone pair genealogy
EIncStChg	The number of inconsistent change of state within the clone pair genealogy
EFltConStChg	The number of re-synchronizing changes (i.e., *RESYNC*) that were a fault fix
EFltIncSChg	The number of diverging changes (i.e., *DIV*) that were a fault fix
EChgTimeInt	The time interval in days since the previous change to the clone pair

We use logistic regression, SVM classifier, Random Forrest, and XGBOOST to classify the clone pairs data. Logistic regression is a statistical model that uses a logistic function to model a binary-dependant variable. Support vector machine (SVM) is a supervised model associated with learning algorithms that analyze data for classification. Random forrest is an ensemble learning method for classification that operates by constructing several decision trees. XGBOOST [14] is an optimized gradient boosting library designed to be highly efficient and flexible. Recent studies [15, 16] have used XGBOOST for training the models for classification problems. We split the data into training (70%) and testing (30%) to train and test the models. We make sure that our data splitting is time consistent i.e., we do not use future late propagations data to predict past late propagations.

Results. Table 8 shows the results of model training using the four machine learning algorithms. We evaluate the models using three performance metrics commonly used for assessing trained machine learning models, including precision, f1-score, and AUC. Precision is the fraction of relevant instances among the retrieved instances. F1-

Table 8 Evaluation metrics for the machine learning algorithms

ML algorithm	M_{LP}			M_{BUG}		
	Precision	F1-score	AUC	Precision	F1-score	AUC
Logistic Regression	0.81	0.68	0.76	0.78	0.71	0.75
SVM Classifier	0.87	0.72	0.80	0.78	0.72	0.76
Random Forrest	0.89	**0.80**	0.85	**0.94**	**0.93**	**0.93**
XGBOOST	**0.91**	0.72	**0.87**	0.91	0.75	0.90

score is the harmonic mean between precision and recall. AUC provides an aggregate measure of performance across all possible classification thresholds. Results show that XGBOOST outperforms all the algorithms in terms of precision and AUC. However, Random Forrest achieves the highest value among the four models.

Furthermore, we analyze the most important predictors for both behaviors (i.e., late propagation occurrence and fault occurrence in late propagation). For M_{LP}, the number of consistent state changes (EConStChg) (37.5%), the number of consistent changes (EConChg) (32%), and the sum of added or changed lines (Tchurn) (23.2%) are the most significant features having more than 90% effect in the model. The number of consistent state changes (EConStChg) has a negative effect, meaning that if a genealogy experience more inconsistent changes than consistent changes, then it can be an indicator of late propagation introduction in clone genealogies. For M_{BUG}, number of fault-prone modifications in the history (CFltRate) (65%), number of previous commits by a specific developer (CommitterExp) (17%), and time interval in days since last change (EChgTime) (8%) are the most significant features having more than 90% effect in the model. The number of faulty changes divided by the number of changes (CFltRate) has a positive effect. A higher number of erroneous changes in clone genealogy history is an indicator of future fault occurrences. Experience has a negative effect, which suggests that late propagation genealogies changed by less experienced developers are more fault-prone. Developers can benefit from these results as they can leverage the trained machine learning models to assess the risks of the clone pairs.

Summary of RQ4

For M_{LP}, XGBOOST achieves the highest precision (0.91) and AUC (0.87) with consistent state changes (EConStChg) being the most significant feature. For M_{BUG}, Random Forrest achieves the highest precision (0.94) and AUC (0.93) with the number of past fault-fixing changes (CFltRate) being the most significant feature.

4 Threats to Validity

We now discuss the threats to the validity of our study. *Construct validity* threats in this study are mainly due to measurement errors possibly introduced by our chosen clone detection tool. To reduce the possibility of misclassification of code fragment as clones, we chose the best configuration for clone detection tool that has been recommended by the recent evaluation of code clone tools [7]. Another construct validity threat stems from the SZZ heuristics used to identify fault-fixing changes [11]. Although this heuristic does not achieve a 100% accuracy, it has been successfully employed and reported to achieve good results in multiple studies [17]. *Reliability validity* threats concern the possibility of replicating this study.[4] We attempt to provide all the details needed to replicate our study. Also, the source code and git repositories of the studied systems are publicly available.

5 Conclusion

In this chapter, we replicate a previous study by Barbour et al. [1] to examine late propagation in more detail. We first confirm the conclusion from the previous study that late propagation is more risky than other clone genealogies. We then identify eight types of late propagation and study them in detail to identify which types of late propagation contribute the most to faults in the systems. Overall, we find that two types of late propagation (i.e., LP7 and LP8) are riskier than the others, in terms of their fault-proneness and the magnitude of their contribution toward faults. LP7 occurs when both clones are modified, causing a divergence and then at least one of the two clones in the pair is modified to re-synchronize the clone pair. LP8 involves no propagation at all and occurs when a clone diverges and then re-synchronizes itself without changes to the other clone in a clone pair. The contribution of other types of late propagation is found to be system-dependent. From this study, we can conclude that late propagation types are not equally risky. We train machine learning models to identify the clone genealogies with late propagation (M_{LP}) and fault-prone late propagations (M_{BUG}) early on. We use 18 different clone genealogy-related features to train four different machine learning models. For the occurrence of late propagation (M_{LP}), XGBOOST achieves the highest precision (0.91) and AUC (0.87) with consistent state changes (EConStChg) being the most significant feature. For the fault-prone late propagations (M_{BUG}), Random Forrest achieves the highest precision (0.94) and AUC (0.93) with the number of fault-prone changes (CFltRate) being the most significant feature.

[4] https://github.com/qecelab/latepropagation.

References

1. L. Barbour, F. Khomh, Y. Zou, Late propagation in software clones, in *2011 27th IEEE International Conference on Software Maintenance (ICSM)* (IEEE, 2011), pp. 273–282
2. M. Kim, V. Sazawal, D. Notkin, G. Murphy, An empirical study of code clone genealogies, in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13 (ACM, New York, NY, USA, 2005), pp. 187–196
3. L. Aversano, L. Cerulo, M. Di Penta, How clones are maintained: an empirical study, in *11th European Conference on Software Maintenance and Reengineering* (2007), pp. 81 –90
4. S. Thummalapenta, L. Cerulo, L. Aversano, M. Di Penta, An empirical study on the maintenance of source code clones. Empir. Softw. Eng. **15**, 1–34 (2010)
5. L. Barbour, L. An, F. Khomh, Y. Zou, S. Wang, An investigation of the fault-proneness of clone evolutionary patterns. Softw. Qual. J. **26**(4), 1187–1222 (2018)
6. N. Göde, R. Koschke, Incremental clone detection, in *13th European Conference on Software Maintenance and Reengineering* (IEEE, 2009), pp. 219–228
7. J. Svajlenko, C.K. Roy, Evaluating modern clone detection tools, in *2014 IEEE International Conference on Software Maintenance and Evolution* (IEEE, 2014), pp. 321–330
8. N. Göde, Evolution of type-1 clones, in *Proceedings of the 9th International Working Conference on Source Code Analysis and Manipulation* (IEEE Computer Society, 2009), pp. 77–86
9. J. Krinke, A study of consistent and inconsistent changes to code clones, in *Working Conference on Reverse Engineering* (2007), pp. 170–178
10. C.C.S., whatthepatch—python's third party patch parsing library Online. Accessed 17 Aug 2020
11. J. Śliwerski, T. Zimmermann, A. Zeller, When do changes induce fixes? ACM Sigsoft Softw. Eng. Notes **30**(4), 1–5 (2005)
12. M. Fischer, M. Pinzger, H. Gall, Populating a release history database from version control and bug tracking systems, in *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings* (IEEE, 2003), pp. 23–32
13. D. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*, 4th ed. (Chapman & Hall, 2007)
14. T. Chen, C. Guestrin, Xgboost: a scalable tree boosting system, in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2016), pp. 785–794
15. Z. Chen, F. Jiang, Y. Cheng, X. Gu, W. Liu, J. Peng, XGBoost classifier for DDoS attack detection and analysis in SDN-based cloud, in *IEEE International Conference on Big Data and Smart Computing (bigcomp)* (IEEE, 2018), pp. 251–256
16. S.S. Dhaliwal, A.-A. Nahid, R. Abbas, Effective intrusion detection system using xgboost. Information **9**(7), 149 (2018)
17. M. Abidi, M.S. Rahman, M. Openja, F. Khomh, Are multi-language design smells fault-prone? An empirical study