An Empirical Study on the Fault-Proneness of Clone Migration in Clone Genealogies

Shuai Xie[1], Foutse Khomh[2], Ying Zou[1], Iman Keivanloo[1]
[1] Department of Electrical and Computer Engineering, Queen's University, Canada.
{shuai.xie, ying.zou, iman.keivanloo}@queensu.ca
[2] SWAT, Polytechnique Montréal, QC, Canada.
foutse.khomh@polymtl.ca

***Abstract*—Copy and paste activities create clone groups in software systems. The evolution of a clone group across the history of a software system is termed as clone genealogy. During the evolution of a clone group, developers may change the location of the code fragments in the clone group. The type of the clone group may also change (*e.g.*, from Type-1 to Type-2). These two phenomena have been referred to as *clone migration* and *clone mutation* respectively. Previous studies have found that clone migration occur frequently in software systems, and suggested that clone migration can induce faults in a software system. In this paper, we examine how clone migration phenomena affect the risk for faults in clone segments, clone groups, and clone genealogies from three long-lived software systems JBOSS, APACHE-ANT, and ARGOUML. Results show that: (1) migrated clone segments, clone groups, and clone genealogies are not equally fault-prone; (2) when a clone mutation occurs during a clone migration, the risk for faults in the migrated clone is increased; (3) migrating a clone that was not changed for a longer period of time is risky.**

***Index Terms*—Type of clones; clone genealogy; clone migration; clone mutation; fault-proneness.**

I. INTRODUCTION

When two or more code segments have a certain similarity or are exactly the same, they are considered to be code clones. Based on the textual similarity among code segments, clones can be classified into four types [1]:

- **Type-1:** Identical code segments except for variations in whitespace, layout and comments.
- **Type-2:** Syntactically identical segments except for variations in identifiers, literals, types, whitespace, layout and comments.
- **Type-3:** Copied segments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout or comments.
- **Type-4:** Code segments implemented through different syntactic variants that perform the same computation.

Code clones are usually introduced in software systems by developers, inadvertently or through copy and paste activities. When more than one duplication is made from a code segment, several clones are created. These clones form a clone group (also known as clone class). A clone group is composed of multiple cloned segments. Duplications of code segments by developers can cross distant parts of a software system and span multiple revisions of the system. Hence, increasing the risk for faults since developers may forget to propagate some of the changes performed on one clone to other clones in the clone group.

The evolution of a clone group during the revisions of a software system is named clone genealogy. A clone genealogy captures the creation, the modification, and the removal of clone groups during the revisions of a software system. *Clone mutation* [2] refers to the changes in clone types (*e.g.*, a Type-1 clone becoming a Type-2 clone after some modifications), while a *clone migration* occurs when a revision changes the location in the source code directory structure of a cloned code.

Many researchers have examined how clone genealogies affect software quality [3], [4]. In particular, in our previous work [2], we investigated how *clone migration* affects the fault-proneness of clone genealogies, through an analysis of the evolution trend of the distances between clone pairs in a clone group. The results showed that *clone migration* is a risky phenomenon that affects a high proportion (68% of clone groups experienced a migration in ArgoUML) of clones in software systems. If all *clone migrations* in ArgoUML are considered equally prone to faults, this means that 68% of all clone groups must be monitored for faults, which is resource intensive. Because developers have limited resources, they are more interested in identifying which clones are most at risk of faults so that they could be a target for testing and reviews.

In this paper, we study the characteristics of different *clone migration* and estimate the likelihood of faults. Our goal is to identify risky patterns of *clone migration* (eventually associated with *clone mutation*) in order to raise the awareness of developers about risky modifications of clones in their software system. We perform our study on three large open source software systems written in JAVA, *i.e.*, JBOSS, APACHE-ANT, and ARGOUML (the same systems used in our previous work [2]). We address the following three research questions:

RQ1: Are clone genealogies that experienced a clone migration more fault-prone than other clone genealogies?

We categorize *clone migrations* based on the types of clones at three levels of granularity: cloned segments, clone groups, and clone genealogies. We also categorize clone groups and clone genealogies based on the frequency of their *clone migrations*. All migrated clones are identified at the clone

context (*i.e.*, the cloned segment level of granularity). At the clone group (respectively clone genealogy) context, we examine how often *clone migration* affects the fault-proneness of clone groups (respectively clone genealogies). We observe that *clone migration* occurs in clones, clone groups, and clone genealogies, and increases their fault-proneness. Clone groups with a large proportion of migrated clones are more fault-prone than clone groups with smaller proportions of migrated clones.

RQ2: Are clone migrations associated with a mutation more fault-prone than other clone migrations?

We analyze whether *clone migrations* associated with *clone mutation* are more fault-prone than *clone migrations* without *clone mutation*. We also study if different types of *clone mutations* (*e.g.*, from Type-1 to Type-2) during *clone migrations* impact the fault-proneness of clones in clone genealogies differently. Results show that clones experiencing both *clone migration* and *clone mutation* are more fault-prone than clones that experienced only *clone migration*. Moreover, *loose migrations* where the similarity of clones is decreased (*e.g.*, mutation from Type-1 to Type-2) are found to be more fault-prone in two of our three subject systems. The risk for faults is the highest when a clone is mutated from or to a Type-3 clone during *clone migration*.

RQ3: Does the time interval between the migrating change and the last change before migration affect the fault-proneness?

We investigate how the length of time interval between the migration activity and the last change before the *clone migration* affects the risk of having fault in a genealogy. We divide the time interval into different period levels and compare the fault-proneness at different time periods. We observe that a longer time interval between the migrating change and the last change before the *clone migration* increases the risk for faults in the migrated clones.

The rest of the paper is organized as follows. Section II discusses the related literature on code clone and clone genealogies. Section III explains the experiment process of our study. Section IV introduces the approach for each research question and analyzes the results. Section V discusses possible threats to the validity of our study. Finally, Section VI concludes the paper and outlines some avenues for future work.

II. RELATED WORK

Kim *et al.* [5] perform the first study on clone genealogies. They create a tool to generate clone genealogies. Using two JAVA systems and the CCFINDER clone detection tool, they investigate if eliminating clones using refactoring can solve issues related to the fault-proneness of code clones. They conclude that refactoring clones is not helpful when dealing with long-lived and consistently changing clones. In our study, we use a different approach to generate clone genealogies. Our work focus on understanding the evolution of three types of clones experiencing different *clone migration* patterns. We investigate faulty *clone migration* patterns to help developers reduce their maintenance efforts.

Bakota *et al.* [6] show that the changes in similarity of clones can be used to identify smells in code clone evolution. Barbour *et al.* [4], [7] examine the fault-proneness of different clone evolutionary patterns in software systems. They investigate risky states and transitions along the evolution history of clones. They also build models to predict faults in code clones using some genealogy based metrics. Comparing to their study, our work uses the same method to process data and a similar approach to generate clone genealogies. However, we use a different clone detection tool and examine a different aspect of clone genealogies, *i.e.*, *clone migration* in clone genealogies.

Aversano *et al.* [8] investigate clone genealogies in two JAVA open source systems to understand how clones are maintained. They use the SIMSCAN tool to detect the clones and define some patterns of clone evolution. They observe that most clone classes are always maintained consistently and that late propagation genealogies are risky. Using the same method as in [8], Thummalapenta *et al.* [3] performed a study about clone genealogies on four open source software systems written in C and JAVA. They found that clones in a late propagation genealogy are more fault-prone than other clones. In our study, we examine *clone migration*. These studies are limited to Type-1 and Type-2 clones while our work investigates also clone genealogies containing Type-3 clones.

Göde [9] presents an approach to build a model of clone evolution based on source code changes between two versions. He performed an empirical study of the evolution of Type-1 clones using nine open-source systems. He concluded that the ratio of clones is reduced during the lifetime of a software system and that cloned segments exist in systems for more than a year on average. Our study builds clone genealogies of Type-1, 2, and 3 clones by mapping clones at the revision level while Göde [9] maps clones at the version level. Our study also takes advantage of defect information to study fault-proneness of genealogies while Göde [9] only studied the inconsistent changes.

Duala-Ekoko *et al.* [10] present an approach to track clones in evolving software systems. They propose the CLONETRACKER framework which is based on the concept of clone region descriptors (CRDs). CLONETRACKER processes outputs of the SIMSCAN clone detection tool and enables tracking clone regions so that developers can edit the clones. They perform a case study and conclude that CLONETRACKER can track the vast majority of the 3,275 clone regions in the systems. Their study can help developers identify clone regions in software systems, while our study focuses on *clone migration*.

In our previous study [2], we observed that *clone migration* and *clone mutation* occur frequently in clone genealogies. Furthermore, we studied that specific behaviours in *clone migration* and *clone mutation* can increase fault-proneness in clone genealogies. Motivated by the outcome of our previous study [2], in this paper, we continue our research by focusing

on *clone migration* fault-proneness. Specifically, we examine *clone migration* from three different aspects to identify the specific risky behaviour.

III. STUDY DESIGN

This section presents the design of our case study. The *goal* of this study is to evaluate the risk of introducing faults in clones, clone groups and clone genealogies when *migrating* different types of clones. The *motivation* of this study is to inform developers about the increased maintenance effort and cost that results from migrating and mutating clones during the evolution of a software system. Our findings may benefit developers and testers who perform development and maintenance activities during the evolution of a software system. Indeed, developers need to estimate their efforts to make changes, while testers are required to know which code segments should be tested in priority. Development teams could also make use of our results to better assess the risk caused by some *clone migrations* and better focus their testing and reviews.

A. Data Collection

At first, we process the change history of our three subject systems, *i.e.*, JBOSS, APACHE ANT, and ARGOUML. All three systems are written in JAVA. The three systems are from different domains and have different sizes (in terms of lines of code). Table I shows some descriptive statistics about the systems where Processed LOC refers to the total size of the data considered for genealogy extraction and clone detection.

Table I
STATISTIC OVERVIEW OF THE SUBJECT SYSTEMS

System	# LOC	# Proc. LOC	# Revisions	# Genealogies
JBoss	635K	1.6M	109K	1.7K
Apache-Ant	254K	2.3M	10K	23 K
ArgoUML	247K	3.1M	18K	15.6K

JBOSS is an open source application server written in JAVA. JBOSS is a division of Red Hat created in 1999. JBOSS has 109K revisions and about 1.7M LOC (lines of code). We conduct our experimental study on the code snapshots from April 2000 to December 2010.

APACHE-ANT is an open source tool to compile, assemble, test and run applications written in JAVA, C, and C++. APACHE-ANT is written in JAVA and contains 1.0M revisions and over 2.3M LOC in its history. The system was built in January 2000 and is still active today. We study code snapshots from January 2000 to November 2010.

ARGOUML is a open source UML-modeling application. It allows users to model systems, generate the corresponding code skeletons, and reverse-engineer diagrams. ARGOUML was started in January 1998 and is still evolving today. ARGOUML is written in JAVA and has 18K revisions and over 3.1M LOC. We analyze code snapshots of the period from January 1998 to November 2010 covering all of the subsystems and packages available in the versioning control. ARGOUML is used in previous studies on clone evolution [8], [4].

B. Processing Data

We follow the same procedures as our previous work [2] to process data and build clone genealogies. An overview of our approach to process data is shown in Figure 1. We use J-REX [11] to mine the source code repository of each of the three JAVA subject systems. J-REX can identify the revisions that have code changes and output the snapshots of the files that are changed at those revisions. In the next step, we remove all test files and use the NICAD clone detection tool to detect clones on the three systems. In addition to the test cases, we also exclude auto-generated code by discarding code fragment with more than 1000 lines of code. Then we use the clones created at different time periods to build clone genealogies. Finally, we extract all the *clone migrations* in the clone genealogies. We identify the different migration patterns for each research question. For **RQ2**, we also extract *clone migrations* associated with *clone mutation*. More details about our experiment are discussed in the following five sections.

1) Identifying Faults: Similar to the approach in our previous work [2], we use J-REX to extract snapshots of all the revisions in the software systems. We flag all the methods that are modified during a revision. J-REX also helps us to analyze each commit message to identify fault fixes. J-REX uses the heuristics proposed by Mockus *et al.* [12] to identify fault fixing changes. J-REX is reported to have an accuracy for 87% [4]. A previous empirical study performed by Hassan [13] shows that J-REX is comparable to professional developers when identifying fault fixes. The correlation between J-REX and those developers is found to be larger than 0.8.

2) Detecting Clones: The clone detection tool used in this work is NICAD [14]. NICAD is a flexible TXL-based hybrid language-sensitive and text comparison software. It can handle many languages, *i.e.*, C, C-SHARP, JAVA, PYTHON and WSDL. Roy *et al.* [14], claim that NICAD can detect both exact (*i.e.*, Type-1) and near-miss (*i.e.*, Type-2 and Type-3) clones with high accuracy. NICAD has a short processing time and a low memory consumption. We select NICAD because we need to parse all the revisions of each of our studied software system in one shot. NICAD does not require the complete compilation unit. However, each code fragment (*i.e.*, method block) must be syntactically correct.

Before performing clone detection, we remove test files since they are primarily used to test the functions of the software. Same as Barbour *et al.* [4], we remove all the files containing the keyword "test" in their filenames or folder-names and manually verify the removed files. After removing test files, we extract methods from all the remaining files using the extension function in J-REX. We save each method snapshot into a single file and use the NICAD clone detection tool to detect cloned code in the files.

The NICAD clone detection tool has been already used in some previous studies on clone genealogies (*e.g.*, Zibran *et al.* [15]). We use the same parameters and configuration for

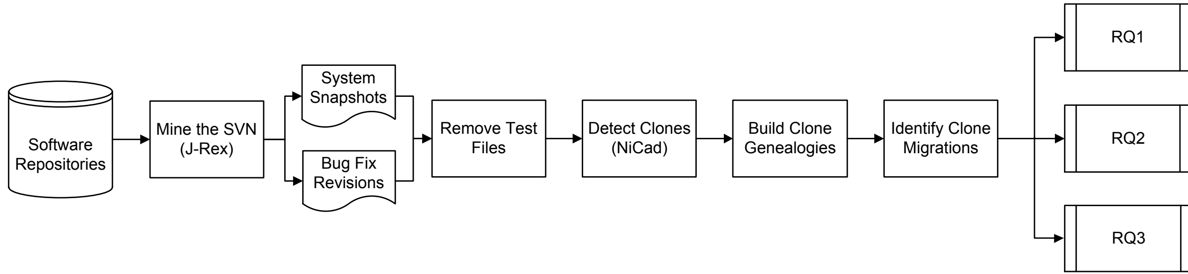

Figure 1. Overview of the Analysis Process

NICAD as in the work of Zibran *et al.* [15]. Table II shows the parameters of NICAD that are used in our study. We use the version 3.4 of NICAD, which is the latest version at the time of this study. We process the results of the clone detection to identify clones that co-exist within the same revision. A similar processing step is done in the work by Barbour *et al.* [4], [16].

Table II
NICAD'S PARAMETERS

Clone Types	Identifier Renaming	Similarity Threshold
Type-1	None	100%
Type-2	Blind-rename	100%
Type-3	Blind-rename	80%

3) Building Clone Genealogies: We build clone genealogies following the same method as our previous work [2]. First, we assign a unique identifier to each version of a code fragment. We create the search space by including all of the identified revisions of all of the code fragments within the given time span. We use NICAD to detect any possible clone pair within the search space. Second, we remove all unchanged clones by mapping clone detection results with the output from the version control systems. In our approach, it is possible to have invalid clone groups in the context of clone genealogies since NICAD is not sensitive to the revision information. We remove the invalid clone groups, *e.g.*, groups containing only code segments belonging to the same method but different revisions. After this pre-processing, we map all of the clones obtained from NICAD across the revisions of the software system. We follow a similar approach as Barbour *et al.* [4] to generate the genealogies of the clone groups. First, we process the output of J-REX to extract the date of each change, for each of the changed methods. We check each modified method to see if the contained clones are modified. We repeat the entire process for each revision. Finally, we extract valid clone groups from the results of the clone detection and link each two of them throughout the revisions of the software system to generate the clone genealogies. To build the clone genealogy, we connect each two clone groups by identifying shared clones, start date and end date for each group.

4) Identifying Clone Migrations: We follow a different approach to track *clone migration* in comparison with our previous study [2]. We follow a different approach since we do not require the distance information. As a result, both approaches are consistent but just capturing different aspects of the migration. We identify *clone migrations* by checking clones with changed location (in the source code directory structure) but unchanged file name. We also examine the order of each two clones to identify the source and destination of each clone migration. Then, we track fault fixes in all of the post migration revisions (*i.e.*, after the migrating action) of migrated clones. For **RQ2**, we also check for *clone mutation* (*i.e.*, changes of clone types) in all *clone migrations*.

C. Statistical Analysis Method

We use the Chi-Square test [17] to check for associations between *clone migration* patterns and future faults. We compute odds ratio (OR) [17] to compare the fault-proneness of different *clone migration* patterns. OR is the ratio of the odds of an event occurring in an experimental group to the odds of it occurring in a control group. OR is computed by $OR = \frac{p/(1-p)}{q/(1-q)}$, where p is the probability of the event occurring in the experimental group and q the probability of it occurring in the control group. An OR value of 1 means that the event is equally likely in both groups. $OR > 1$ means that the event is more likely to occur in the experimental group, while $OR < 1$ indicates that the event is more likely to occur in the control group. We use the 5% level (*i.e.*, p-value <0.05) as the threshold value to identify if the results of the Chi-square test are significant.

IV. CASE STUDY RESULTS

This section presents the results of our three research questions. For each question, we discuss the motivation behind the question, the analysis approach and the findings.

RQ1: Are clone genealogies that experienced a clone migration more fault-prone than other clone genealogies?

Motivation. Our previous study [2] has provided quantitative evidence of the frequent occurrence of *clone migration* in clone genealogies from the three subject systems investigated in this study. However, *clone migration* can be observed in different contexts: clones, clone groups, and clone genealogies. In this question, we want to study the impact of *clone migration* with regard to fault-proneness for these three contexts.

Table III
CATEGORIES OF CLONE GENEALOGIES

Categories	Clone types in the genealogy
G<1>	Type-1
G<2>	Type-2
G<3>	Type-3
G<1,2,3>	Type-1, Type-2, Type-3

In particular, we are interested in understanding if migrated clones are more faulty than non-migrated clones in the context of a clone itself. We also want to understand if the proportion of migrated clones in a clone group would affect the risk for faults when modifying the clones in the clone groups. Moreover, we examine the effect of *clone migration* over the evolution of clone groups (*i.e.*, clone genealogies). The result of this research question will enable developers to better estimate the efforts and the risks related to *clone migration* (*i.e.*, changing the location of one or more clones). This question is preliminary to **RQ2** and **RQ3**, which identify more migration patterns from two different aspects.

Approach. Before identifying *clone migration* in clone genealogies, we classify clone genealogies into four categories as presented in Table III. We use these four categories of clone genealogies to identify how *clone migration* affects the fault-proneness in different clone contexts, *i.e.*, clones, clone groups, and clone genealogies. These categories of clone genealogies are characterized using all the clone types involved in the genealogies. For example, G<1> represents clone genealogies containing only Type-1 clones that remained Type-1 clones all through their history, while the G<1,2,3> category represents genealogies containing all three types of clones.

In the context of clones, we identify migrated clones and examine how *clone migration* directly affects the fault-proneness of code clones. In the context of clone groups, there are different proportions of migrated clones in clone groups, or no *clone migration* at all. In this context we examine how the proportion of migrated clones in a clone group affects the fault-proneness of the clone group. In the context of clone genealogies, there are also different proportions of migrated clones among different clone genealogies. Considering different numbers of clone groups in clone genealogies, we compute the *migration density* to measure the frequency of the *clone migration* that occurred in the clone genealogies. We want to understand if clone groups and clone genealogies will be more fault-prone when they have more *clone migrations*.

In order to examine the fault-proneness of clone genealogies with different numbers of *clone migrations*, we need to categorize the genealogies based on the different proportion levels of *clone migration*. We use the *migration density* of clone genealogies to measure how often *clone migrations* occurred in clone genealogies. Because there are different numbers of clone groups in clone genealogies, we use the number of groups and the number of *clone migrations* to calculate the *migration density*. We compute the *migration density* by $\frac{N_m}{N_g-1}$, where N_m is the number of clone groups experiencing at least one *clone migration* and the denominator corresponds to the number of modifications of the clone group during the evolution of the system within the clone genealogy.

Next, we divide the clone genealogies into five levels based on different proportions of the maximum *migration density* from all the clone genealogies. Then we can identify how different *migration densities* affect the risk for faults in clone genealogies. As shown in the first two rows of Table IV, five levels are identified (from level 0 to level 4). Level 0 stands for genealogies without *clone migration*, level 1 for very low migration frequency, level 2 for low migration frequency, level 3 for high migration frequency, and level 4 for very high migration frequency. In the third row of Table IV, D stands for the *migration density*. Level 4 refers to the largest *migration density* compared with all other genealogies. Similar to clone genealogies, we also divide the clone groups into five levels based on the proportion of migrated clones in each clone group. As shown in the last row of Table IV, P refers to the proportion of migrated clone in a clone group. Therefore, clone groups in level 0 have no migration, *i.e.*, they have a 0 percentage of migrated clone. Clone groups in level 4 (very high migration) have the largest proportion of migrated clones.

We perform the Chi-square test at a 5% level for p-value to verify if *clone migrations* at different levels are related to a higher risk for fault. We also compute odds ratios respectively for migrated clones, different proportion levels of *clone migrations* in clone groups and different density levels of migrated clones in clone genealogies. We categorize the migration-containing clone groups into four experimental groups based on the proportion of migrated clones as introduced in IV. We also divide the migration-containing genealogies into four experimental groups by computing the *migration density*. The control groups for all three contexts are respectively clones, clone groups, and clone genealogies without *clone migration*. To mitigate the potential impact of the similarity threshold used to detect Type-3 clones, on our results, we perform the computation using six different similarity thresholds (*i.e.*, 70%, 75%, 85%, 90% and 95%) for G<3> and G<1,2,3> introduced in III, for all three subject systems.

We address **RQ1** by building clone genealogies and identifying *clone migration* in three subject systems following the methods described in Section III. We examine the fault-proneness of migrated clones, clone groups and clone genealogies that contain migrated clones. Based on this question, we formulate the following null hypothesis: H_1*: Clone migration does not affect the fault-proneness of clones, clone groups, and clone genealogies.*

Findings. Table V shows odds ratio results for migration-containing clones, clone groups, and clone genealogies. The following three paragraphs discuss odds ratios results for clone, clone group, and clone genealogy contexts.

Clone Context: We compare the fault-proneness between migrated clones and non-migrated clones by comparing the odds ratios. The fault flag "1" means migrated clones have larger odds ratios and hence are more fault-prone than non-migrated clones. The fault flag "0" means the opposite. The "p-value" column of clone context shows that one value in APACHE-

Table IV
MIGRATION DENSITY OF CLONE GENEALOGIES AND MIGRATION PROPORTION OF CLONE GROUPS

Names	No Migration	Very Low Migration	Low Migration	High Migration	Very High Migration
Levels	0	1	2	3	4
Migration Density of Clone Genealogies	0	$0<D \leqslant 0.25$	$0.25<D \leqslant 0.5$	$0.5<D \leqslant 0.75$	$0.75<D \leqslant 1$
Migration Proportion of Clone Groups	0	$0<P \leqslant 0.25$	$0.25<P \leqslant 0.5$	$0.5<P \leqslant 0.75$	$0.75<P \leqslant 1$

Table V
FAULT-PRONENESS OF CLONE MIGRATIONS IN CLONES, CLONE GROUPS, AND CLONE GENEALOGIES

Contexts		Clones						Clone Groups						Clone Genealogies					
System		JBoss		Apache-Ant		ArgoUML		JBoss		Apache-Ant		ArgoUML		JBoss		Apache-Ant		ArgoUML	
Clone Type	Similarity	Fault Flag	P-Value	Fault Flag	P-Value	Fault Flag	P-Value	Proportion Level	P-Value	Proportion Level	P-Value	Proportion Level	P-Value	Proportion Level	P-Value	Proportion Level	P-Value	Proportion Level	P-Value
G<1>	100%	1	**<0.05**	1	0.831	1	0.205	0	0.152	0	**<0.05**	0	**<0.05**	0	**<0.05**	0	**<0.05**	4	**<0.05**
G<2>	100%	1	**<0.05**	1	**<0.05**	0	**<0.05**	2	**<0.05**	2	**<0.05**	3	**<0.05**	1	**<0.05**	0	**<0.05**	1	**<0.05**
G<3>	70%	1	**<0.05**	1	**<0.05**	0	**<0.05**	4	**<0.05**	4	**<0.05**	2	**<0.05**	4	0.370	1	**<0.05**	0	**<0.05**
	75%	1	**<0.05**	1	**<0.05**	1	**<0.05**	4	**<0.05**	4	**<0.05**	3	**<0.05**	1	**<0.05**	2	0.156	1	**<0.05**
	80%	1	**<0.05**	0	**<0.05**	1	**<0.05**	4	**<0.05**	4	**<0.05**	3	**<0.05**	1	**<0.05**	0	**<0.05**	1	**<0.05**
	85%	1	**<0.05**	0	**<0.05**	1	0.082	2	**<0.05**	0	**<0.05**	2	**<0.05**	1	**<0.05**	3	0.164	1	**<0.05**
	90%	1	**<0.05**	0	**<0.05**	1	**<0.05**	2	**<0.05**	4	**<0.05**	2	**<0.05**	2	**<0.05**	3	**<0.05**	1	**<0.05**
	95%	1	**<0.05**	0	**<0.05**	1	**<0.05**	2	**<0.05**	4	**<0.05**	2	**<0.05**	0	**<0.05**	1	0.109	1	**<0.05**
G<1, 2, 3>	70%	1	**<0.05**	1	**<0.05**	0	**<0.05**	4	0.601	0	**<0.05**	0	**<0.05**	1	**<0.05**	2	**<0.05**	0	**<0.05**
	75%	1	**<0.05**	1	**<0.05**	0	**<0.05**	4	**<0.05**	0	**<0.05**	3	**<0.05**	1	**<0.05**	2	**<0.05**	3	**<0.05**
	80%	1	**<0.05**	1	**<0.05**	0	**<0.05**	4	**<0.05**	0	**<0.05**	3	**<0.05**	3	**<0.05**	1	**<0.05**	4	**<0.05**
	85%	1	**<0.05**	1	**<0.05**	0	**<0.05**	4	**<0.05**	0	**<0.05**	3	**<0.05**	1	**<0.05**	0	**<0.05**	2	**<0.05**
	90%	1	**<0.05**	0	**<0.05**	0	**<0.05**	4	**<0.05**	2	**<0.05**	3	**<0.05**	4	**<0.05**	1	**<0.05**	3	**<0.05**
	95%	1	**<0.05**	0	**<0.05**	0	0.459	2	**<0.05**	0	**<0.05**	3	**<0.05**	1	**<0.05**	2	0.132	2	**<0.05**

ANT and three values in ARGOUML are larger than our Chi-square test threshold (0.05). Overall, the results for clone context are statistically significant. However, both APACHE-ANT and ARGOUML show inconsistent behaviours such as Type-3 (G<3>) clones in APACHE-ANT and clones with three types of clones (G<1,2,3>) in ARGOUML. Therefore, we cannot conclude that migration in the clone context is a generalizable indicator for fault-proneness.

Clone Group Context: The "proportion level" column in Table V presents the most faulty proportion level. Clone groups containing *clone migration* are more faulty if the value is larger than 0, while a larger value means that groups with larger proportions of migrated clones are more fault-prone than others. Only two values in JBOSS are larger than our Chi-square test threshold (0.05), thus the overall result for the clone group context is statistically significant. We find that most of the values are 3 and 4, which means that clone groups with larger percentage of migrated clones are more fault-prone. Therefore, from the results of the three subject systems, we can conclude that clone groups are more faulty when they have migrated clones, and having a larger proportion of migrated clones increases the risk for future faults. However, in all three subject systems, Type-1 (G<1>) clone groups without *clone migration* are more fault-prone.

Clone Genealogy Context: The "proportion level" column for genealogy context in Table V presents the most faulty proportion level. The proportion level measures how large is the *migration density* in a clone genealogy. A high proportion level refers to high *migration density* in comparison to other clone genealogies. As shown in Table V, there is only one value in APACHE-ANT and four values in ARGOUML that are larger than our Chi-square test threshold (0.05), hence the overall result for clone genealogies are statistically significant. Since most of the values are larger than 0, which means that no *clone migration* occurred in the clone genealogies, we conclude that clone genealogies with *clone migrations* are more faulty excluding Type-1 (G<1>) clone genealogies. As we notice Type-1 related observations are not completely consistent with the rest of the study, we explore this issue in the following discussion section.

Results for the other five similarity thresholds, for all three contexts, in all three subject systems, are statistically significant. This means that fault-proneness degree is different between migration-containing and non-migration-containing clones, clone groups, and clone genealogies. Overall, we can reject H_1. Based on our results, we suggest that developers pay more attention when migrating near-miss clones.

Discussion: In this section, we observed that the migration event in the context of clone group and genealogy is a reliable indicator for fault-proneness. However, Type-1 clones are the exceptions in our study. Similar to the results of the clone group context, Type-1 (*i.e.*, G<1>) clone genealogies without *clone migration* are more fault-prone in JBOSS and APACHE-ANT. In order to find an explanation behind this exceptional

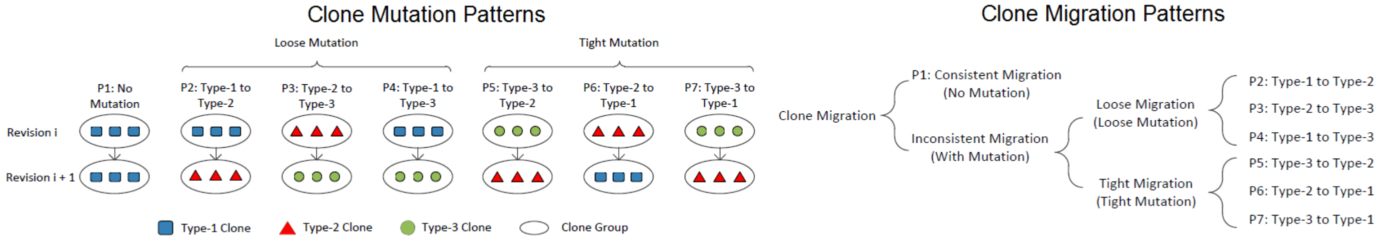

Figure 2. The Clone Mutation and Clone Migration Patterns

behaviour, we examine the source code of Type-1 clones in our subject systems. We also examine the potential impact of LOC (lines of code) on our results by comparing the average LOC of all migrated clones in the clone genealogies for the three systems. The results show that the average LOC for faulty and non-faulty migrated clones are respectively, 32 and 33 in JBOSS, 48 and 43 in APACHE-ANT, and 25 and 24 in ARGOUML. Therefore, there is no significant difference between the LOC of faulty and non-faulty migrated clones. We conclude the LOC of the migrated clone has no impact on our results shown in Table V. Concerning Type-1 clones, we examined the source code of every Type-1 clone pair and observed that whenever a Type-1 clone pair exists, both pairs always have identical method name. We also notice that the Type-1 cloned methods are always within either the same class or sibling classes. This behaviour is significantly different from Type-2 and 3 clones. Such coarse grained similarity decreases the chance of inconsistent change in the cloned code. Inconsistent change in code clones is a known factor to the fault-proneness of the cloned code [5]. This observation may explain why Type-1 clones with migration behave differently (*i.e.*, not significantly more fault-prone or even less faulty) from the other clone types in our study in Table V.

Overall, we conclude that clones, clone groups, and clone genealogies containing *clone migration* have higher fault-proneness in software systems.

In the first research question, we observed that *clone migration* in genealogies is related to higher fault-proneness. In the next research questions, we study this phenomena (*i.e.*, *clone migration*) into more details to identify if some migrations (Figure 2) are more fault-prone than others.

RQ2: Are clone migrations associated with a mutation more fault-prone than other clone migrations?

Motivation. When making *clone migration* for clone segments, the clone type could be changed due to code changes on the cloned code segment. We refer to changes on clone type as *clone mutation*. We refer to the *consistent* migration as the *clone migration* without changing the clone type. We identify a set of *clone migration* patterns along with different *clone mutations* on clone types (Figure 2). We aim to verify whether the existence of *clone mutation* in the *clone migration* will affect the fault-proneness of the migrated clones in the future period. Identifying the effect of code changes causing *clone mutation* in *clone migration* can help developers decide the risk of changing and moving cloned code.

Approach. When the clone type is changed after *clone migration*, the change on clone type can make the clones have higher or lower similarity. We define the phenomena of changing clone types to the one with a higher similarity as *tight mutation* and to the one with a lower similarity as *loose mutation*. The *loose mutation* leads to a change on clone type from a lower type with a higher similarity to a higher type with a lower similarity. While the *tight mutation* leads to a higher similarity between clones. We flag our seven migration patterns from P1 to P7, where the *consistent* pattern with no *clone mutation* is flagged as P1. The *loose migration* pattern contains *loose mutation* and the *tight migration* pattern contains *tight mutation*. For *loose migration* pattern, we define three migration patterns with all change possibilities among three types of clones. Figure 2 shows the examples for the seven migration patterns. For example, a code clone changes clone type from Type-1 to Type-2 in the P2 *loose mutation* pattern. We define three related migration patterns (*i.e.*, P2, P3, and P4) for *loose mutation* and define other three related migration patterns (*i.e.*, P5, P6, and P7) for *tight mutation*. As shown in Figure 2, those six migration patterns (*i.e.*, P2 to P7) present all the possible mutation scenarios among the three clone types.

For each subject system, we build clone genealogies following the approach introduced in Section III-B3. Next, we identify the *clone migrations* and classify them based on the different mutation patterns, which are shown in Figure 2. For all seven migration patterns, we compute the number of fault-containing and fault-free genealogies and formulate the following null hypothesis: H_2*: The clone migrations with and without clone mutation are equally fault-prone.*

We detect Type-3 clones with a selected similarity threshold of 80% (see Table II). To assess the potential impact of this chosen threshold on our results, we perform our study by repeating the detection of Type-3 clones using other similarity thresholds, *i.e.*, 70%, 75%, 85%, 90% and 95%. For each of these similarity thresholds, we build clone genealogies, classify them into seven migration patterns presented in Figure 2 with different mutation patterns. We repeat the testing of H_2 using the Chi-square test and odds ratios.

We use all seven migration patterns (ie P1 to P7) in Figure 2

Table VI
ODDS RATIOS OF CLONE MIGRATION PATTERNS ALONG WITH CLONE MUTATION

System	Type-3 Similarity	P1: No Mutation	P2: Type-1 to Type-2	P3: Type-2 to Type-3	P4: Type-1 to Type-3	P5: Type-3 to Type-2	P6: Type-2 to Type-1	P7: Type-3 to Type-1	P-Value
JBoss	70%	1	**2.67**	1.19	0.71	0.93	**1.38**	0.69	<**0.05**
	75%	1	**2.09**	1.07	0.66	0.82	**1.08**	0.67	<**0.05**
	80%	1	2.11	**7.39**	2.66	**2.37**	0.78	1.1	<**0.05**
	85%	1	1.87	**9.36**	3.06	**2.72**	0.69	1.36	<**0.05**
	90%	1	1.5	**7.5**	4.14	**10.91**	0.55	4.64	<**0.05**
	95%	1	1.12	0	**4.73**	-	0.44	-	<**0.05**
Apache -Ant	70%	1	1.66	**1.99**	1.53	**1.81**	1.2	1.62	<**0.05**
	75%	1	1.37	**1.71**	1.63	1.08	0.92	**1.38**	<**0.05**
	80%	1	1.48	**1.97**	1.74	**1.35**	0.94	1.32	<**0.05**
	85%	1	1.37	**2.16**	1.94	**1.5**	0.89	1.35	<**0.05**
	90%	1	**1.77**	1.12	1.42	1.19	1.22	**1.86**	<**0.05**
	95%	1	1.29	1.93	**2.77**	1.18	0.91	**2.97**	<**0.05**
Argo -UML	70%	1	0.35	3.11	**5.43**	5.2	1.12	**8.43**	<**0.05**
	75%	1	1.1	9.2	**18.01**	13.55	3.53	**25.89**	<**0.05**
	80%	1	1.58	9.5	**24.2**	24.84	5.07	**48.49**	<**0.05**
	85%	1	1.77	16.51	**37.76**	31.12	5.69	**83.4**	<**0.05**
	90%	1	2.19	14.41	**27.97**	37.83	7.02	**122.48**	<**0.05**
	95%	1	2.62	0	**122.29**	0	8.42	**88.05**	<**0.05**

to compute odds ratios. When computing odds ratios, we select the *consistent* migration pattern (P1) without *clone mutation* as the control group. We build six experimental groups containing clones that experienced patterns P2 to P7 respectively. We perform the Chi-square test using the 5% level.

Findings. Table VI shows the results of the Chi-square test and odds ratios for the *clone migration* without *clone mutation*, with different *tight migration* and *loose migration* patterns. The results present six similarities for G<1,2,3> category. The *clone migration* without *clone mutation* on the clone types is the control group, thus they all have a value 1. The largest value for each experimental group of migration pattern (*i.e.*, *loose mutation* and *tight migration*) is highlighted in bold.

From the results, we find that the results of migration pattern with *loose mutation* (*i.e.*, P2, P3, or P4) have higher odds ratios than non-mutation *clone migration* and *tight migration* patterns (*i.e.*, P5, P6, or P7) for JBOSS and APACHE-ANT except for the result of 90% similarity in JBOSS, 90% and 95% similarity in APACHE-ANT. Thus for JBOSS and APACHE-ANT, the *loose migration* patterns (*i.e.*, P2, P3, or P4) changing the clone type from the one with higher similarity to the one with a lower similarity, are more fault-prone than other patterns. While for ARGOUML, except the results for 95% similarity, cloned code that experienced *tight migration* patterns (*i.e.*, P5, P6, or P7) have higher fault-proneness than the other patterns.

Moreover, in Table VI, migrated clones with *clone mutation* between Type-2 and Type-3 (*i.e.*, Type-2 to Type-3 (P3) and Type-3 to Type-2 (P5)) are more fault-prone for the results of about more than half the similarity thresholds in both JBOSS and APACHE-ANT systems. *Clone migration* containing *clone mutation* from Type-3 to Type-1 (P7) are also more fault-prone in APACHE-ANT. While for ARGOUML, *clone migration* with changing clone type from Type-3 to Type-1 (P7) is the most fault-prone one for five in six similarity thresholds. *Clone migration* from Type-2 to Type-1 (*i.e.*, P6) includes most of our exceptional cases within these observations. As discussed earlier, our inspection revealed that this different behaviour is due to the special characteristics of Type-1 clones in the subject systems, which are inherently having less inconsistent changes (*i.e.*, a common source of fault-proneness in cloned code [5]).

All of the results shown in Table VI are statistically significant. Overall, based on the results we get, we can reject H_2. We conclude that *loose migration* is more fault-prone in two of the three systems and *clone mutation* involving Type-3 clone makes the *clone migration* more fault-prone in both directions (*e.g.*, P3, P4, and P5).

Overall, we conclude that migrated clones containing *clone mutation* have higher risk for faults.

RQ3: Does the time interval between the migrating change and the last change before migration affect the fault-proneness?

Motivation. *Clone migration* can be carried out immediately after the last change made to the cloned code segment or in a long period after the change. A long time interval between the *clone migration* and the last change to the clone code segment may lead to a higher chance to introduce defects when making a *clone migration*. We examine this question to help developers learn about the risk of introducing defects when applying *clone migration* by considering the length of time interval after the last code change before *clone migration*.

Approach. We compute the number of clones that have faults and have no faults for *clone migrations* with different time intervals between the last code change and *clone migration*. We use 200 days as the unit to divide the length values into different levels (*i.e.*, $\frac{N_p}{200}$, N_p is the number of days between the last code change and *clone migration*). Level value of 1

means that the migration occurred after 200 days from the last change and lower values means that migrations occured within 200 days. We select 200 days as our unit of time, however, using other units will provide levels that are isomorphic to the ones presented in this paper, and therefore will yield equivalent results. The three systems have the different lengths of evolution, thus three subject systems have different levels. We perform this method on six similarities (*i.e.*, 70%, 75%, 80%, 85%, 90% and 95%) for G<1,2,3> category and the four different categories of clones presented in Table III.

We compute the Chi-square test and the odds ratio to examine the following hypothesis: H_3*: the length of the time interval between the last code change and clone migration will not affect the number of faults in the migrated clones*.

Finding. As presented in Table VII and VIII, odds ratios in different period levels are calculated considering the entire history of each system. Level 0 means the *clone migration* is made in the shortest time period after the last code change, while the largest level that has valid odds ratios refers to the longest observed period.

The results for cloned code in clone genealogies containing all three types of clones are presented in Table VII. In the JBOSS system, the highest odds ratios for all similarity thresholds are presented in period level 8 (from 1601 to 1800 days), which is the ninth period level of thirteen period levels. For APACHE-ANT system, the most fault-prone period levels are 9 (from 1801 to 2000 days), 14 (from 2801 to 3000 days), 15 (from 3001 to 3200 days), and 16 (3201 to 3400 days) out of 17 period levels for different similarity thresholds. While in ARGOUML, the period level 5 (from 1001 to 1200 days) out of 12 period levels is most fault-prone for five of six similarity thresholds. The only exception is period level 7 (from 1401 to 1600 days) that has the most fault-prone result 70% similarity. Overall, we can conclude that the *clone migration* is more fault-prone if the time interval between the last change and migration becomes larger than half of the history of the software system.

Moreover, Table VIII shows the results for four different categories of clone genealogies defined in Table III from **RQ1**. There is no valid odds ratio for cloned code in Type-1 clone genealogy for JBOSS, this is due to few *clone migration* experiences and the special characteristics of the Type-1 clones that are discussed earlier. In these results, we find that in Type-3 clone genealogies (G<3>) and clone genealogies with all three clone types (G<1,2,3>), migrated clones made in a longer interval time than the middle period level are more fault-prone for three systems, except in G<1,2,3> of ARGOUML. For *clone migrations* in Type-1 clone genealogies (G<1>), the higher period level (*i.e.*, 8 out in 17) has the highest risk for faults in two of three systems.

All the P-value results shown in Table VII and Table VIII are larger than 0.05, therefore they are statistically significant and we can reject H_3. In general, we conclude that Type-3 related clone genealogies that are involved in *clone migrations* occurring in a longer time interval (since the last code changes) are more fault-prone than the ones made after a shorter time interval.

Overall, we conclude that the *clone migration* made in a longer time interval after the last code change is riskier.

V. THREATS TO VALIDITY

In this section, we analyze the threats to validity for this study. We follow the common guidelines [18] for empirical studies.

Construct validity threats are related to the relation between theory and observation. In this study these threats mainly result from the reliability of the tool that we use to detect code clones. We use NICAD to detect clones since it detects both exact and near-miss code clones with enough precision and recall [14]. It is a stable clone detection tool and is also used in our previous study [2]. Note, NiCad is limited to block and method-level clone detection.

The other threat for construct validity is the accuracy of J-REX, which uses the same algorithm as previous studies from Hassan *et al.* [19] and Mockus *et al.* [20]. Hassan [13] has conducted an experiment to compare a classification of commit messages with a manual evaluation of commit messages by six professional developers, and found a correlation of $\sigma > 0.8$. Thus, the ability to recognize bug fixes of J-REX is proved to be comparable to the ability of professional developers.

There is no threat to *internal validity* in our study, which is an exploratory study [18]. Even though we try to explain the observations in our analyses, we cannot claim causation. We just report observations and correlations from our results.

Conclusion validity threats concern the relation between the treatment and result. We carefully analyze the assumptions for the statistical test. We use non-parametric tests without making assumptions on the distribution of data.

Reliability validity threats deal with the possibility to repeat this empirical study. All the three subject systems used in our study have available data for public. We also reported the configurations of the third-party tools used in this study such as NICAD and J-REX and the details of our experiments.

Threats for *external validity* are about how to generalize our results. Each of three large open source software systems used in this study has a plug-in architecture. They are written in the same language (JAVA), still, they represents different domains and various project sizes (line of code). Nevertheless, more analysis using more subject systems are desirable.

VI. CONCLUSION

In this study, we examine how *clone migration* affects the risk for faults in software systems. We identify the faults of *clone migration* on three contexts, *i.e.*, clones, clone groups, and clone genealogies. Our results show that clone genealogies with *clone migration* have higher fault-proneness for the clones in all three contexts (*i.e.*, clones, clone groups, and clone genealogies). Specifically, our observation highlights that the frequency of migration in a clone genealogy has a direct correlation with the fault-proneness of the genealogy.

Furthermore, we investigate the fault-proneness of the *clone migration* without *clone mutation*, *loose migration* and *tight*

Table VII

ODDS RATIOS OF CLONE MIGRATIONS FOR DIFFERENT PERIOD LEVELS IN TYPE-3 CLONE GENEALOGIES

Days Ranges		0-200	201-400	401-600	601-800	801-1000	1001-1200	1201-1400	1401-1600	1601-1800	1801-2000	2001-2200	2201-2400	2401-2600	2601-2800	2801-3000	3001-3200	3201-3400	P-Value
G<1,2,3>	Similarity	Level 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
Jboss	70%	1	0.95	0.93	0.57	0.89	1.87	0.9	0	**107.13**	1.28	0	-	1.28	-	-	-	-	<**0.05**
	75%	1	1.06	1.44	1.8	0.92	0.69	1.54	0	**116.08**	1.66	0	-	2.76	-	-	-	-	<**0.05**
	80%	1	0.64	1.22	1.46	0.44	0	0.92	0	**137.13**	0.65	-	-	0	-	-	-	-	<**0.05**
	85%	1	0.59	1.18	1.83	0.42	0	0.91	-	**118.87**	0	-	-	-	-	-	-	-	<**0.05**
	90%	1	0.66	1.02	1.33	0.56	0	0.73	-	**90.55**	0	-	-	-	-	-	-	-	<**0.05**
	95%	1	0.7	0.73	1.15	0.18	0	0	-	**1.74**	0	-	-	-	-	-	-	-	<**0.05**
Apache-Ant	70%	1	1.09	1.32	0.77	0.39	0.69	1.11	3.03	2.15	1.96	0.71	5.59	0.83	1	20.89	5.99	**39.52**	<**0.05**
	75%	1	1	0.84	0.89	0.33	0.62	0.83	0.67	0.71	2.84	0.18	6.81	6.82	0.29	23.07	**86.75**	28.92	<**0.05**
	80%	1	0.85	1.11	0.51	0.25	0.42	0.85	0.56	0.4	161.89	0.26	2.98	4.6	0.87	52.22	**167.11**	0	<**0.05**
	85%	1	0.93	1.22	0.6	0.22	0.48	0.29	0.58	0.42	143.61	0.3	4.2	0.17	-	**51.29**	-	0	<**0.05**
	90%	1	1.54	0.23	0.13	0.06	0.27	0.24	0.12	0.14	**147.84**	0.52	0	-	-	-	-	-	<**0.05**
	95%	1	1.8	0.37	0.25	0.11	0.32	0.3	0.22	0.13	**121.53**	0.4	0	-	-	-	-	-	<**0.05**
ArgoUML	70%	1	8.65	2.1	1.02	2.96	33.11	57.5	**186.49**	21.97	11.11	8.37	0.85	0	-	-	-	-	<**0.05**
	75%	1	11.46	1.64	1.83	7.03	**241.88**	68.63	145.04	69.35	37.68	9.23	0	0	-	-	-	-	<**0.05**
	80%	1	11.13	1.5	1.24	5.48	**374.38**	80.69	204.92	67.73	57.2	2.19	0	0	-	-	-	-	<**0.05**
	85%	1	11.69	1.64	1.3	6.71	**614.82**	118.05	281.71	67.92	118.25	2.86	0	0	-	-	-	-	<**0.05**
	90%	1	11.72	1.2	1.3	7.97	**777.94**	147.66	327.93	86.44	149.18	3.68	0	0	-	-	-	-	<**0.05**
	95%	1	13.3	1.27	1.06	7.52	**964.93**	161.65	322.71	36.34	168.85	4.5	0	0	-	-	-	-	<**0.05**

Table VIII

ODDS RATIOS OF CLONE MIGRATIONS FOR DIFFERENT PERIOD LEVELS IN FOUR CATEGORIES OF CLONE GENEALOGIES

Days Ranges			0-200	201-400	401-600	601-800	801-1000	1001-1200	1201-1400	1401-1600	1601-1800	1801-2000	2001-2200	2201-2400	2401-2600	2601-2800	2801-3000	3001-3200	3201-3400	P-Value
System	Category	Similarity	Level 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
JBoss	G<1>	100%	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	G<2>	100%	1	1.48	0.48	**3.41**	0	-	0	-	1.56	-	-	-	-	-	-	-	-	<**0.05**
	G<3>	80%	1	0.51	1.02	1.81	1.2	0	**3.61**	0	-	1.81	-	-	0	-	-	-	-	<**0.05**
	G<1,2,3>	80%	1	0.64	1.22	1.46	0.44	0	0.92	0	**137.13**	0.65	-	-	0	-	-	-	-	<**0.05**
Apache-Ant	G<1>	100%	1	0.96	0.57	0	0	1.6	2.86	0	**6.67**	-	3.33	-	-	-	-	-	-	<**0.05**
	G<2>	100%	1	2.52	0.4	0.17	0	0	0	**18.75**	0	-	3.13	0	-	-	-	-	-	<**0.05**
	G<3>	80%	1	0.44	1.15	0.57	0.21	0.34	0.57	0.71	0.29	71.54	0.04	2.34	4.87	1.04	22.59	**203.31**	0	<**0.05**
	G<1,2,3>	80%	1	0.85	1.11	0.51	0.25	0.42	0.85	0.56	0.4	161.89	0.26	2.98	4.6	0.87	52.22	**167.11**	0	<**0.05**
ArgoUML	G<1>	100%	1	0.21	0.08	0	0	0.76	1.21	0	**1.52**	1.18	0	0	-	-	-	-	-	<**0.05**
	G<2>	100%	1	12.86	1.58	0.96	3.54	**3117.74**	249.42	683.5	88.97	383.6	11.27	0	-	-	-	-	-	<**0.05**
	G<3>	80%	1	1.31	0.37	0.34	0.36	4.12	0.72	4.68	**9.73**	0.15	0.1	0	0	-	-	-	-	<**0.05**
	G<1,2,3>	80%	1	11.13	1.5	1.24	5.48	**374.38**	80.69	204.92	67.73	57.2	2.19	0	0	-	-	-	-	<**0.05**

migration respectively. We also examine the impact on the risk for faults in *clone migration* of six different mutation patterns with changing clone types. We find that the existence of *clone mutation* makes the *clone migration* riskier. Between the *loose migration* and *tight migration*, we find that *loose migrations* that reduce the similarity between clones is more fault-prone in two of the three subject systems. We also found that *clone mutations* involving Type-3 (*i.e.*, between Type-1 and Type-3, between Type-2 and Type-3) make the *clone migrations* more fault-prone for all the three systems.

Finally, we examine how the length of time interval between the *clone migration* change and the last change made on these clones affects the risk of *clone migration* in terms of fault-proneness. The results show that, a longer time interval between *clone migration* and the last change yields a significantly higher fault-proneness for the migrated clones related to Type-3 and Type-1. In the future, we aim to extend our study by examining more software systems written in other programming languages.

REFERENCES

[1] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, “Comparison and evaluation of clone detection tools,” *IEEE Trans. Software Eng.*, pp. 577–591, 2007.

[2] S. Xie, F. Khomh, and Y. Zou, “An empirical study of the fault-proneness of clone mutation and clone migration,” in *MSR*, 2013, pp. 149–158.

[3] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta, “An empirical study on the maintenance of source code clones,” *Empirical Software Engineering*, vol. 15, pp. 1–34, 2010.

[4] L. Barbour, F. Khomh, and Y. Zou, “Late propagation in software clones,” in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, sept. 2011, pp. 273 – 282.

[5] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy, “An empirical study of code clone genealogies,” in *ESEC/SIGSOFT FSE’05*, 2005, pp. 187–196.

[6] T. Bakota, R. Ferenc, and T. Gyimothy, “Clone smells in software evolution,” in *IEEE International Conference on Software Maintenance*, 2007, pp. 24 –33.

[7] L. J. Barbour, “Empirical studies of code clone genealogies,” Master’s thesis, Depart of Electrical and Computer Engineering, Queen’s University, Kingston, Ontraio, Canada, 2012.

[8] L. Aversano, L. Cerulo, and M. Di Penta, “How clones are maintained: An empirical study,” in *11th European Conference on Software Maintenance and Reengineering*, 2007, pp. 81 –90.

[9] N. Göde, “Evolution of type-1 clones,” in *SCAM’09*, 2009, pp. 77–86.

[10] E. Duala-Ekoko and M. P. Robillard, “Tracking code clones in evolving software,” in *ICSE’07*, 2007, pp. 158–167.

[11] W. Shang, Z. M. Jiang, B. Adams, and A. Hassan, “Mapreduce as a general framework to support research in mining software repositories (msr),” in *6th IEEE International Working Conference on Mining Software Repositories*, May 2009, pp. 21 –30.

[12] A. Mockus and L. Votta, “Identifying reasons for software changes using historic databases,” in *Proceedings. International Conference on Software Maintenance*, 2000.

[13] A. E. Hassan, “Predicting faults using the complexity of code changes,” in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 78–88. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070510

[14] C. K. Roy and J. R. Cordy, “Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization,” in *ICPC’08*, 2008, pp. 172–181.

[15] M. F. Zibran, R. K. Saha, M. Asaduzzaman, and C. K. Roy, “Analyzing and forecasting near-miss clones in evolving software: An empirical study,” in *ICECCS’11*, 2011, pp. 295–304.

[16] G. M. Selim, L. Barbour, W. Shang, B. Adams, A. E. Hassan, and Y. Zou, “Studying the impact of clones on software defects,” *Working Conference on Reverse Engineering*, pp. 13–21, 2010.

[17] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures, Fourth Edition*. Chapman & Hall/CRC, Jan. 2007.

[18] R. K. Yin, “Design and methods third edition, 3rd ed.” in *ICSM’00*, 2002.

[19] A. E. Hassan and R. C. Holt, “Studying the evolution of software systems using evolutionary code extractors,” in *IWPSE’04*, 2004, pp. 76–81.

[20] A. Mockus and L. G. Votta, “Identifying reasons for software changes using historic databases,” in *ICSM’00*, 2000, pp. 120–130.