Studying the Impact of Clones on Software Defects

Gehan M. K. Selim, Liliane Barbour, Weiyi Shang, Bram Adams, Ahmed E. Hassan, Ying Zou
Queen's University
Kingston, Canada
gehan@cs.queensu.ca, l.barbour@queensu.ca, {swy, bram, ahmed}@cs.queensu.ca, ying.zou@queensu.ca

Abstract—There are numerous studies that examine whether or not cloned code is harmful to software systems. Yet, few of them study which characteristics of cloned code in particular lead to software defects. In our work, we use survival analysis to understand the impact of clones on software defects and to determine the characteristics of cloned code that have the highest impact on software defects. Our survival models express the risk of defects in terms of basic predictors inherent to the code (e.g., LOC) and cloning predictors (e.g., number of clone siblings). We perform a case study using two clone detection tools on two large, long-lived systems using survival analysis. We determine that the defect-proneness of cloned methods is specific to the system under study and that more resources should be directed towards methods with a longer 'commit history'.

I. INTRODUCTION

Code clones are traditionally defined as segments of code that are highly similar or identical in terms of their semantics or structure. Two or more clones form a clone class, and the members of the class are known as clone siblings. Clones can be introduced intentionally (e.g., copy and paste behavior) or unintentionally in both the development and maintenance phases of the software lifecycle.

Previous empirical studies have tried to establish a causality relationship between clones and software defects. Some researchers claim that clones reduce the maintainability of code. They argue that inconsistent changes to clones within a clone class can introduce software defects [7]. For example, if a developer makes a change to a method, but is unaware of clone siblings of that method, the fix will not be applied to the clones, possibly leading to defects. Those against cloning also argue that 'blind' copy and pasting of code can lead to software defects if the developer does not understand how the pasted code will interact with existing code. However, opposing studies claim that there are circumstances in which clones are justified. In those cases, clone management techniques are required instead of removing the clones through refactoring [6, 14]. For example, clones might be used to maintain the stability of a system by avoiding unstable experimental code containing defects.

Typically, limited resources are available for inspecting and testing code, so it is unreasonable to verify that each clone is defect-free. Our goal is to provide managers with a set of predictors that can be used to identify which clones are most at risk of containing a defect. This information can be used to focus code testing efforts.

In this paper, we study the impact of various clone characteristics using two subject systems and two clone detection tools. We use survival models to calculate the risk of an event occurring (e.g., a defect) over time, given certain predictors. We build hazard and survival models to address two research questions:

RQ1: *Can we model the impact of clones on defects with high accuracy?*
We determine if the relationship between clones and defects can be generalized across different systems. We also examine if cloned code is overall more or less defect-prone than non-cloned code.

RQ2: *What are the most important predictors of defects in cloned code?*
Previous research has tried to classify clones in general as either helpful or harmful to a software system. Filtering the clones based on defect-prone predictors can help focus code testing and review efforts.

The rest of this paper is organized as follows. Section 2 provides an overview of survival and hazard models. Section 3 explains our study design. Section 4 describes the study results. Section 5 lists some threats to the validity of the study. Section 6 summarizes related empirical studies on code clones. Finally, Section 7 summarizes and concludes the paper.

II. COX HAZARD MODELS

In this study, we analyze method revisions to study the relationship between cloned method revisions and defects. We also determine the characteristics of cloned method revisions that have the highest impact on defects. In regression modeling each subject, in this case each method, can correspond to only one observation (i.e., row) in the data. However, since each method can have more than one revision, we need a way to represent each method as several observations in the data.

Cox hazard models model the instantaneous risk (or 'hazard') of the occurrence of an event as a function of a set of predictors [2, 10, 11]. In particular, Cox models are used in survival analysis to model how long subjects under observation 'survive' before the occurrence of an event. For example, Fox [11] used Cox models to represent the risk of recently released prisoners being rearrested, where the 'rearrest' is the event of interest.

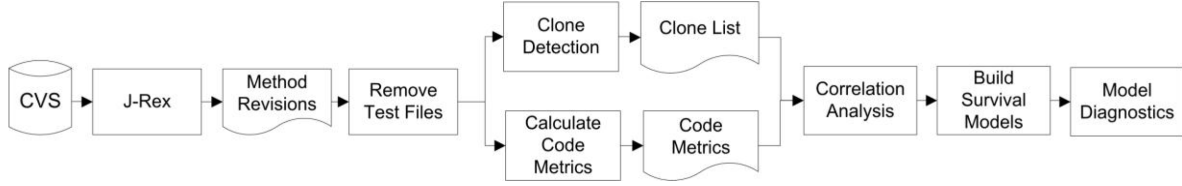

Figure 1. Overview of our approach

Unlike regression models, Cox models allow each subject to have time dependent covariates. Each subject is reflected in the data as multiple observations over time. Each observation includes the start and the end time of the observation, a flag signifying the occurrence of the event of interest, and a set of covariates. Hence, we used Cox models rather than standard regression models in order to model the 'risk' of a method experiencing an event over time. The occurrence of a defect is the event of interest in our models. The subjects in this study are methods, and each observation (i.e., one row) corresponds to one method revision. Since a method can experience a defect more than once throughout its lifetime, we use an extended version of standard Cox models that can handle 'recurrent events'. In other words, such Cox models can handle data in which the event of interest (e.g., occurrence of a defect) can occur more than once for a method.

The hazard or probability of experiencing a defect at time t is modeled by the following hazard function:

$$\lambda_i(t) = \lambda_0(t) * e^{\beta * X_i(t)} \quad (1)$$

Or equivalently, taking the log of both sides, we get

$$log(\lambda_i(t)) = log(\lambda_0(t)) + \beta_1 x_{i1}(t) + \ldots + \beta_k x_{ik}(t) \quad (2)$$

where:

- $X_i(t)$ is the vector of time-dependent predictors of observation *i* at time t
- β is the vector of coefficients for the predictors in $X_i(t)$
- λ_0 is the baseline hazard
- *k* is the number of predictors

The baseline hazard can be thought of as the hazard of occurrence of the event of interest when all the predictors have zero effect on the hazard. The baseline hazard is cancelled out when calculating the relative hazard between two classes (i.e. two methods in our case) at a specific time, as shown in equation (3) below [2].

$$\lambda_i / \lambda_j = e^{\beta(x_i(t) - x_j(t))} \quad (3)$$

This implies that the relative hazard is a function of only the predictors' values, not of the baseline hazard. This assumption is referred to as proportional hazards assumption. The proportional hazards assumption states that the effect of predictors is stable over time and that the effect of the predictor does not show a trend (e.g., increase or decrease) with time. A diagnostic test has to be carried out to check that this assumption is satisfied, hence validating the Cox model. Other diagnostic tests were proposed in the literature [2,11], but only the proportional hazards diagnostic test was carried out in this study.

Equation (2) shows that the log of the hazard is a linear function of the log of the baseline hazard and all the predictors taken into consideration. If a linear relationship between the log hazard and the predictors does not exist, a link function should be applied to the predictors to make this linear relationship valid. Link functions are used to transform the predictors to a new scale, hence making the predictors linearly related to the log hazard. Choosing an appropriate link function for predictors in Cox hazard models is essential so that identical changes in the value of a predictor affect the log hazard equally, regardless the original value of the predictor. This allows the Cox models to uphold the proportionality assumption.

The literature proposes several hazard model variants. We selected Cox hazard models for this study for four reasons:

1. Only some of the study subjects (i.e., a method) must experience the event (i.e., a defect).The models allow subjects to leave the study early or survive the entire observation period without experiencing the event.
2. The subjects can be grouped according to the value of one predictor (e.g. clone or non-clone), with each group having an individual baseline hazard within the model [2,11] .
3. The characteristics of the subjects can change over time. For example, the LOC of a method can change each time the method is changed.
4. Cox models have been adapted for recurrent events. Accommodating recurrent events is suitable for modeling open source software development, where software modules continuously evolve over time, and a method can have multiple defect fixes over its lifetime [2].

III. STUDY DESIGN

This section describes the approach we use to populate the code predictors and answer our research questions. A general overview of our approach is shown in Figure 1. We mine the CVS source code repository of each subject system using a tool called J-REX to produce a copy of each method revision and identify method revisions that fix a defect. We then perform clone detection and measure the code metrics (e.g. LOC, cyclomatic complexity) of each method revision.

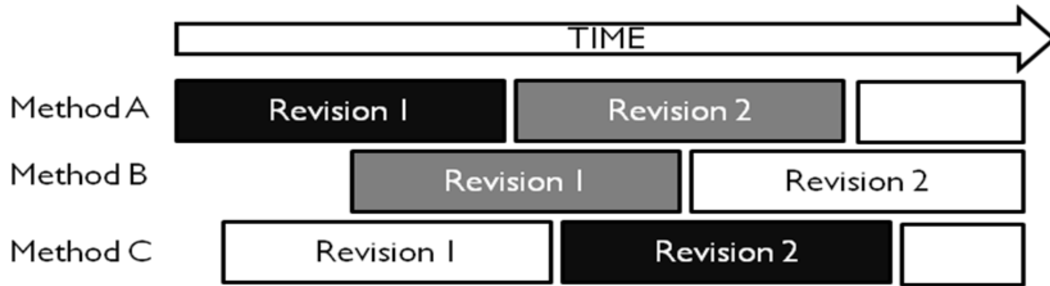

Figure 2. Post processing of clone detection results

Using the survival package in the R tool, we build and validate survival models for each subject system. We describe each step in more detail in the remainder of this section.

A. *Mining the CVS using J-Rex*

Similar to C-REX [1], J-REX is used to study the evolution of source code of Java software systems. For each subject system, we build a list of all methods and their revisions using J-REX. This list is correlated with the bug repository to identify revisions that contain a defect fix. Using the information from the CVS, it is only possibly to identify revisions where a defect fix occurred, but it is unknown which revision introduced the defect [2].

The approach used by J-REX is broken down into three phases:

1. Extraction: J-REX extracts source code snapshots for each Java file revision in a CVS repository.
2. Parsing: Using the Eclipse JDT parser, J-REX builds an abstract syntax tree for each extracted file revision and stores the tree in an XML document.
3. Analysis: J-REX compares the XML documents of consecutive file revisions to determine changed code units and generates evolutionary change data. The results are stored in an XML document. There is one XML document for each Java file.

In addition to the source extraction and evolutionary analysis, J-REX uses a heuristic on all commit log text to determine the reason for the commit. For example, a commit log text containing the word "bug" is assigned the type "bug". We use the same heuristics as proposed by Mockus et al. [3]. Using the assignments in the J-REX output, we identify the method revisions that contain a defect fix.

J-REX takes as input the CVS repository of a Java system. In our study, systems that used a SVN repository were first migrated to CVS before executing J-REX. No data required by our study was lost during conversion, nor was the data modified in a way that would change our results.

B. *Removing Test Files*

The subject systems include test files that are used by the developers to test the project subsystems. Some of these test files contain incomplete code. Test files increase the number of clones, since many test files are copied and then modified slightly to test different cases. We remove such test files from all systems.

C. *Clone Detection*

Past researchers execute clone detection on a subset of the repository. For example, Rahman et al. [9] detect clones between monthly snapshots of the system. Using this technique leads to a loss of information, since the clone status of revisions between the snapshots is lost. This information is important for developers, since the change from clone to non-clone might indicate a defect. The period between snapshots must be selected to minimize the loss of data.

Our approach avoids the use of snapshots. We perform clone detection using all the method revisions, no revisions are skipped. All the method revisions from the entire version history of a subject system are submitted to an existing clone detection tool. Clone detection is performed once to detect clones between all the revisions. This introduces noise, such as clones between method revisions that never co-existed or between revisions of the same method.

More specifically, the contents of each method revision output by J-REX are extracted into individual files (i.e., one file per method revision). To uniquely identify which revision file belongs to a specific method revision, we set the file name as a hash of the method path information and the revision number. Existing clone detection tools can be used without modification to detect clones between the method revision files. After detection, the file name is used to map the clones back to the matching method revision and label the revision as a clone.

Two revisions can only be labeled as clones if they exist simultaneously (co-exist) within the history of the software system. It does not make sense to propagate a change historically to a past revision of a method. An example of post processing is shown in Figure 2. In the figure, revision 1 of method A and revision 2 of method C are identified as a possible clone pair, but removed from the clone list because they never co-exist. Revision 1 of method B and revision 2 of method A are a clone pair because they overlap during their lifetimes. For these reasons, we label a method revision as a clone only under the following conditions:

- the clone detection tool identifies a clone sibling
- the clone sibling co-existed with the revision under study

The validity of our study is dependent on whether the clones identified by the clone detection tool are valid clones. For this reason, we use two different clone detection tools

TABLE I. PREDICTORS USED IN COX MODELING

Predictors	Variable Name	Description
Control Predictors		
Lines of Code	loc	A raw count of the number of lines of code for a method.
Tokens	tokens	CCFinder returns clones as a range of code tokens within a file. The total number of tokens in each method is provided by CCFinder.
Nesting	nesting	The maximum number of nesting levels for a method.
Cyclomatic Complexity	cyclo	The number of if-tests within a method.
Clone	clone	This variable is true if the method contains at least one clone.
Cumulative Defects	culdefects	The number of method defects up to and including the current revision.
Cumulative Defects/ Number of Revisions	numdefectnumprev	The method defect density.
Cloning Characteristic Predictors		
Born Clone	bornclone	True if the first revision of the method is a clone.
Number of Cloned Revisions	numrev	The number of cloned revisions for a method up to and including the current revision.
Number of Clone Siblings	numsib	The total number of clone siblings of a method revision. The siblings could belong to different clone classes.
Number of Defect Siblings	numdefectsib	The total number of clone siblings that contain a defect.
Average Cumulative Defect Siblings	avgculdefectsib	The cumulative number of defects for a method revision is the number of defective revisions for all revisions of the same method up to and including all the current revision. For each cloned revision, we calculate the average across all clone siblings.
Average Normalized Cumulative Defect Siblings	avgculdefectsibavg	The revision number of a clone sibling is not consistent across all siblings. It is possible that one sibling has 4 defects within 8 revisions and another has 4 defects within 12 revisions. To normalize these results, we divide the cumulative number of defects by the revision number of the clone sibling. Then, for each cloned revision, we calculate the average normalized cumulative number of defects across all clone siblings.
Cloning Characteristic Predictors - Simian Specific		
Cloned Lines of Code	cloneLOC	The number of method LOC that are cloned.
Clone Coverage (Simian only)	percentCloneLOC	The percentage of the method that is cloned, based on the total number of LOC.
Cloning Characteristic Predictors - CCFinder Specific		
Clone Tokens	cloneTokens	The number of method tokens that are cloned.
Clone Coverage	percentCloneTokens	The percentage of the method that is cloned, as a percentage of the total number of method tokens.

and compare the results. We conduct our study using CCFinder, a token-based clone detection tool, and Simian, a string-based clone detection tool. Both tools identify clones that are exact matches and clones with minor modifications (e.g., identifiers have been renamed). Neither can detect clones between segments of code where lines of code have been added or removed. These clones are known as "gapped clones".

D. *Gathering Code Metrics*

Based on the method revision files we create for clone detection, we calculate metrics used as predictors in the hazard models, such as Lines of Code (LOC) and cyclomatic complexity. These metrics are described in more detail in Section 3.E.2.

The data collected in all the previous steps is stored in a MySQL database. This allows us to aggregate metrics for each method across multiple revisions.

E. *Building Survival Models*

1) *Identifying Link Functions for Predictors*

We followed a similar approach to the one followed by Koru et al. [2] for identifying a link function for LOC in each considered data set. Like Koru et al., we plot the log relative risk vs. LOC, and visually identify an appropriate link function. For simplicity, we find the optimal link function for the LOC predictor and apply it to all the other predictors. As described in Section 2, a link function ensures that the proportionality assumption for the Cox model is satisfied.

2) *Building Cox Models*

The Cox Models are created using the survival package in R [22]. A summary of the predictors used in this study is shown in Table 1. The predictors are sorted into two categories: control predictors and cloning predictors. For each data set we build two Cox models:

- *Model based on Control Predictors*: This model uses predictors inherent to the method revision, such as LOC, cyclomatic complexity and the number of nesting levels. The control predictors are predictors used in former studies to build models for defect analysis. Using control predictors allows us to build models that can be easily compared to models from other studies.
- *Model based on Control and Cloning Predictors*: This model is used to study the effect of cloning predictors in presence of control predictors. This

TABLE II. SUBJECT SYSTEMS

System	Total LOC	Number of Methods	Number of Method Revisions	CCFinder Clones (%)	Simian Clones (%)	Revisions Containing Defects (%)	Number of Revisions in Study	Number of Revisions in Study (%)
Apache Ant	1.41M	17.57K	61.02K	30.78%	1.99%	11.32%	60.85K	99.72%
ArgoUML	1.76M	23.30K	92.55K	10.57%	1.73%	21.26%	5.68K	6.14%

TABLE III. PREDICTORS ELIMINATED ACCORDING TO CORRELATION ANALYSIS

Data Set	Predictors Eliminated	
	Control Predictors	*Environment Predictors*
ArgoUML Simian	culdefects	cloneLOC, avgCulDefectSib
ArgoUML CCFinder	culdefects	avgCulDefectSib
Ant Simian	nesting, cyclo, culdefects	avgCulDefectSib
Ant CCFinder	nesting, cyclo, tokens, culdefects	cloneToken, percentCloneToken, avgCulDefectSib

model allows us to study the effect of cloning predictors on the defect proneness of methods. Simian outputs clone ranges in LOCs whereas CCFinder outputs clone ranges in tokens. Some of the clone predictors reflect the choice of the clone detection tool. The cloning predictors were selected to capture characteristics of cloned code and clone siblings that are not commonly investigated in former studies. However, our intuition is that characteristics of cloned code and clone siblings can have a major impact on code defects and can give interesting insight as to what profoundly affects the risk of experiencing a defect. For example, it is interesting to learn whether having many clone siblings makes a method more defect-prone since it is harder to consistently maintain many clone siblings as compared to maintaining one or two clone siblings.

Each Cox model was stratified based on whether or not the method revisions are a 'clone'. Stratification generates different baseline hazards for different classes of observations. This stratification sets aside the effect of whether or not a method revision is cloned, and hence makes the effect of other predictors more prominent [2]. In this study, the models have different baseline hazards for cloned and non-cloned method revisions.

F. *Interpretation of Results and Validation of Models*

The R package [20] produces a summary with various statistics of the models, which we examine to interpret the results in terms of the following factors:

- The effect of each predictor (e.g., a predictor's coefficient) on the defect proneness. The sign of a coefficient (e.g., positive or negative) signifies the direction of change in defect proneness (e.g., increase or decrease) when the predictor increases.
- The standard error of each predictor coefficient.
- The correlation between the actual and the predicted number of occurrences of defect fixes. This allows us to quantify how well the models represent the hazard of defect occurrence of the raw data.

TABLE IV. TEST FOR PROPORTIONALITY ASSUMPTION

Data Set	Test for Proportionality Assumption		
	rho	*chisq*	*p-value*
ArgoUML – Simian	0.02690	0.821	0.3650
ArgoUML – CCFinder	0.03070	1.070	0.3000
Ant – Simian	0.02260	3.100	0.0783
Ant – CCFinder	-0.00766	0.372	0.5420

We carry out the proportional hazards diagnostic test for all our models to prove their validity. Other diagnostic tests are available for Cox models [11] and can be investigated as future work to further validate our models.

IV. STUDY RESULTS

A. *Subject Systems*

As our subject systems, we use two open source Java projects: Apache Ant and ArgoUML. They are selected because of their use in previous studies. Table 2 describes the characteristics of the two subject systems. We use two clone detection tools, Simian and CCFinder, on each of the two subject systems. Therefore, we build models for four data sets: Ant-Simian, Ant-CCFinder, ArgoUML-Simian and ArgoUML-CCFinder.

B. *Correlation Results*

For each of the four data sets, correlation analysis between the predictors is carried out to eliminate redundant predictors. Hence, different predictors were eliminated for each data set. We used '0.8' as the correlation threshold over which any two predictors are considered correlated. For each pair of correlated predictors, we eliminate one predictor. The remaining predictors are used to build the hazard models. Table 3 shows for each data set which predictors were eliminated according to the correlation results.

C. *Identifying a link function of the LOC predictor*

We determine the appropriate link function for LOC in each of the four data sets by plotting the log relative risk vs. LOC as done by Koru et al. [2]. We examine the four plots created by the four data sets to find a suitable link function

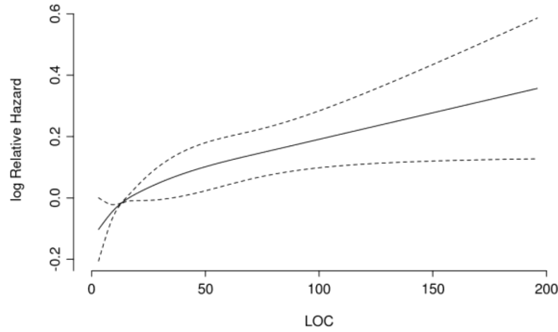

Figure 3. Determining the link function for Ant

for LOC. However, due to the large LOC range, we notice that the plots do not exhibit an obvious functional form throughout the entire range and hence we cannot find one link function that fits the LOC data. Hence, we decide to identify the optimal LOC range for each data set that exhibits a functional form (e.g., shows a pure logarithmic or cubic behavior in this range) in its plot. Since choosing a link function is intended so that the model passes the proportionality assumption test, we ensure that the chosen range of LOC passes the proportionality assumption with a high p-value. Any method revisions with LOC outside the selected range are discarded from the study. The number of method revisions used overall in the study is show in the last column of Table 2.

Each of the two subject systems had a different optimal LOC range. The range is independent of the choice of clone detector. Future work should extend this by building models for the data sets that work across the whole LOC range for all studied systems. The selected LOC range for Ant is 0-200 and for ArgoUML is 55-300. As shown in Table 2, in the case of Ant, almost all the revisions (99.72%) are included in this range. The optimal range for ArgoUML limited the study to only 6% of the revisions.

Figure 3 shows the relationship between LOC and log relative hazard for the Ant-Simian data set after selecting functions within the chosen LOC range. All subject systems showed similar plots for their selected LOC range. This implied that we can use a logarithmic link function for LOC.

D. Building Cox Models

We build two Cox models for each of the four data sets using all predictors except for the predictors removed during the correlation analysis. This results in eight Cox models. After building each model, we examine the summary of the model and iteratively remove insignificant predictors (p>0.05), starting with the variable having the highest p-value. Table 5 shows the set of predictors that were most significant in determining defects in each of the eight models. Most systems showed that LOC and the history of defects of a method (e.g. numdefectnumprev) are control predictors that are the most significant in determining defects. From the cloning predictors, the number of defective clone siblings (e.g. NumDefectSib) is significant in determining defects in most systems.

TABLE V. SIGNIFICANT PREDICTORS IN PREDICTING DEFECTS

Data Set	Model Based on Control Predictors Only	Model Based on Control and Cloning Predictors
ArgoUML Simian	loc numdefectnumprev clone	loc, numdefectnumprev numclonesib, NumDefectSib, clone
ArgoUML CCFinder	tokens, clone numdefectnumprev	tokens, clone numdefectnumprev avgCulSibDefectAvg
Ant Simian	loc, clone numdefectnumprev	loc, numdefectnumprev, numclonerev, clone, numclonesib, NumDefectSib
Ant CCFinder	loc, clone, numdefectnumprev	loc, numdefectnumprev, bornclone, clone, numclonesib, NumDefectSib

For each of the eight models, we generate the summary of the model. Hence, we can interpret the effect of each predictor in each model. For space limitations, we show the summary of the model based on control and cloning predictors for the Ant-CCFinder data set in Table 6. Positive coefficients signify that an increase in the corresponding predictor will increase defects. Therefore, defects increase with an increase in LOC, with an increase in the number of defects in the history of a method (numdefectnumprev) and with an increase in the number of cloned siblings (numCloneSib). However, defects would decrease if a method was born as a clone (bornclone) and with an increase in the number of defective clone siblings (numDefectSib). Table 6 also shows that a control predictor (e.g. numdefectnumprev) can have the highest impact on predicting defects with a coefficient relatively much higher (e.g., 1.145) than that for other predictors and with a very low p value (p-value < 2*10e-16). Overall, control predictors are more important in determining the defect proneness of a method than cloning predictors.

The summaries of all eight models can be interpreted in a similar manner. The signs of the coefficients for the same predictors varied for the different data sets. In the future, we plan to explore more data sets, and reach a general consensus regarding the effect of each predictor on defects and why the effect of the same predictor differs across data sets.

E. Analysis of Models

We structure our discussion of the analysis of the models around our two research questions stated in section I.

RQ1: Can we model the impact of clones on defects with high accuracy?

To determine whether the generated Cox models can represent the impact of clones on defects with high accuracy, we calculate the Spearman correlation between the actual and predicted occurrence of defects according to our models. Table 7 shows the results obtained. All the models show medium to high correlation. Models built for ArgoUML show lower correlation than other data sets due to the low

TABLE VI. SUMMARY OF MODEL BASED ON CONTROL AND CLONING PREDICTORS FOR ANT - CCFINDER

	coef	Se(coef)	z	p
loc	0.102	0.0213	4.8	1.60e-06
numdefectnumprev	1.145	0.066	17.35	<2*10e-16
bornclone	-0.325	0.0517	-6.29	3.21e-10
numclonesib	0.107	0.0293	3.64	2.68e-04
numdefectsib	-0.368	0.0816	-4.51	6.52e-06

TABLE VII. SPEARMAN CORRELATION BETWEEN ACTUAL AND EXPECTED OCCURRENCE OF DEFECTS

	Model based on Control Predictors	Model based on Control and Cloning Predictors
ArgoUML –Simian	0.5956751	0.6099531
ArgoUML - CCFinder	0.5525268	0.5083261
Ant – Simian	0.6809132	0.6424308
Ant –CCFinder	0.7038093	0.5814575

TABLE VIII. P-VALUE OF THE TEST FOR PROPORTIONALITY ASSUMPTION

Data set	Model based on Control predictors only	Model based on Control and Cloning predictors
ArgoUML –Simian	0.468	0.01597
ArgoUML - CCFinder	0.55	0.367
Ant –Simian	0.00823	0.02913
Ant –CCFinder	0.02435	0.1527

number of revisions used from ArgoUML (Table 2). Not all data sets in Table 7 show a higher correlation when the number of predictors used in the models increases. In fact, only one data set (i.e., ArgoUML-Simian) shows a higher correlation when building the Cox model using control and cloning predictors. The three other data sets show a higher correlation when their models are built using only control predictors. This indicates that practitioners can use traditional control predictors to predict defects in systems and still get high prediction results.

To further check the validity of the models, we test the proportionality assumption on the eight models. Table 8 shows the p-value of the test for proportionality assumption for the eight models. This is different from the tests shown in Table 4 because the tests in Table 8 are based on all the predictors, not just LOC. Table 8 reveals that all models pass the test for proportionality assumption (p-value > 0.01) except for the Ant-Simian data set, which did not pass the test using the model based on control predictors only.

For each of the eight models, we plot the survival curves for cloned and non-cloned functions. This helps us determine whether non-cloned functions can survive defects more than cloned functions as traditionally claimed. Due to space limitations, in Figure 4 we only show the survival curves for ArgoUML-Simian and Ant-Simian when using the model based on control and cloning predictors. The CCFinder data sets are similar to the Simian data sets, but with a smaller

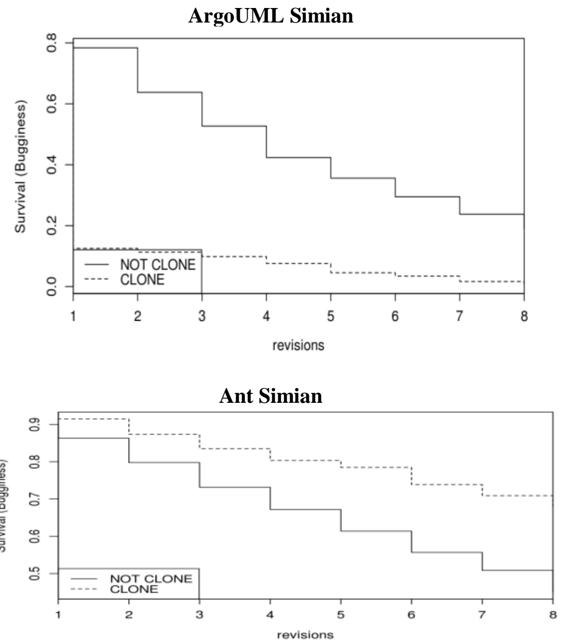

Figure 4. Survival curves of cloned and non cloned methods in ArgoUML-Simian and Ant-Simian using the control and cloning predictor models

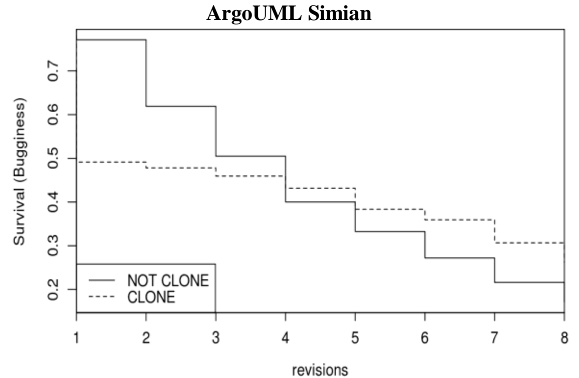

Figure 5. Survival of methods in the ArgoUML-Simian data set, using only the control predictors

gap between the survivals of cloned and non-cloned methods.

In Figure 4, different data sets seem to show different behaviour for cloned and non-cloned methods. The Y axis is the probability of a method surviving defects. Hence low numbers on the Y axis signify a low survival rate (i.e. high hazard or high risk of experiencing defects) For some data sets (e.g. Argouml-simian), non-cloned methods survive defects better than cloned methods since their survival curves have higher values on the Y axis. For such data sets, non-cloned methods are less risky than cloned methods, as traditionally claimed. Especially for the ArgoUML-Simian data set, cloned methods are highly risky and can barely survive the occurrence of defects. On the other hand, the Ant-Simian data set shows that cloned code is safer than

non-cloned code. We deduce that we cannot build a single Cox model to represent different data sets, since different data sets show different survival probabilities for cloned and non-cloned methods.

RQ2: What are the most important predictors of defects in cloned code?

As previously discussed in Section 4.D, each model shows a different set of predictors that are significant in detecting defects. However, from Table 5 we see that most of the models based on control and cloning predictors have numDefectSib as a common significant predictor.

To further explore the effect of using additional cloning predictors we compare two plots. Figure 5 shows the survival curve for ArgoUML-Simian when using the model based on control predictors only and Figure 4 shows the survival curve for ArgoUML-Simian when using the model based on control and cloning predictors. Figure 5 reveals that using only the control predictors, the data set shows that cloned revisions are initially risky in the first four revisions of their lifetime, then become more stable than non-cloned revisions in the second half of their lifetime. However, the corresponding plot for ArgoUML-Simian in Figure 4 shows that cloned code is more risky than non cloned code throughout their lifetime. Hence, incorporating cloning predictors into the model reveal features of methods that are initially not apparent. In case of ArgoUML, non-cloned code survives defects better than cloned code. In the case of Apache Ant, cloned code survives defects better than non-cloned code. This leads to the finding that the stability of cloned code is intrinsic to the system under study.

V. Threats to Validity

A. Identifying Defect Fixes

Our study is based on the data provided by J-REX, a software evolution tool that generates high-level evolutionary change data of the source code of software projects. J-Rex uses heuristics to identify defect-fixing changes [3]. The results of the paper are dependent on the accuracy of the results from J-REX. We are confident in the results from J-REX as it implements the same algorithm used previously by Hassan et al. [1] and Mockus et al. [3].

B. Clone Detection

The classification of a method revision as a clone or non-clone is only as accurate as our choice of clone detection tool. To mitigate the possibly of misclassification, we chose two clone detection tools and repeat the study for both tools.

C. Choice of Link Function

We selected a logarithmic link function for all predictors. It is possible that a different link function would be a better fit for some of the predictors. The diagnostics indicate that the models were a good fit for the data. Thus the choice of link function does not have a significant impact on the model fit. We leave the selection of different link functions for each predictor as future work.

D. Limitation of LOC Ranges

For each subject system, we limit our analysis to a range of LOC values. This reduces the number of revisions used in the study. To minimize this risk, we select two systems. ArgoUML is limited to only 6% of the total revisions, but in the case of Ant, almost all revisions are used in the study. In the future we plan to investigate more systems.

VI. Related Work

A. Predicting Failure Proneness in Software Systems

Koru et al. [2] explore the relationship between the size of a Java class and its risk of defects. They build Cox models of defect fixes using class size as the only predictor. The study provides evidence that small classes are proportionally more defect prone than large classes. Cataldo et al. [15] investigate the effect of different kinds of software dependencies on the probability of observing faults. Different dependency measures are defined and used to build a logistic regression model to predict failure proneness at the file level. All dependencies prove to increase failure proneness, with differences in their degree of impact.

Mockus et al. [4] build models to predict customer perceived software quality based on a set of predictors. The results show that the deployment time, the hardware configuration, and the software platform are vital factors in forecasting the probability of observing a software failure.

Zimmermann et al. [23] construct the Eclipse bug data set and show that complexity has a high positive correlation with failure.

B. Tracking Evolution of Clones in Software Systems

Harder and Göde [12] discuss the available clone evolution models and corresponding techniques that track clones between versions of a software system. The study describes three major clone evolution patterns, the clone detection approaches used, the clone-mapping methodologies, and their evaluation.

Kim and Notkin [17] propose a clone genealogy extractor that generates the clone history of code. The study defines a clone evolution model. Results show that many clones change inconsistently. Hence, maintaining clones could be a better solution than refactoring.

C. Studying Failure Proneness as a Function of Code Cloning

Juergens et al. [8] propose an algorithm to track inconsistently changing clones. The authors then investigate bugs related to such clones. Results show a precision of up to 38% in detecting bugs related to inconsistent clone changes. Aversano et al. [13] further investigate co-change on clones and whether or not a bug fix effort is propagated to a clone class. A case study showed that clones can be maintained to a high degree if the co-change happens within a small time window.

Bettenburg et al. [18] argue that studying clone evolution and maintenance at the release level would provide insight into the effects of code cloning as perceived by customers. A

case study proves that at most 3% of inconsistent changes to clones at the release level lead to bugs.

Rahman et al. [9] investigate the effect of cloning on defect proneness. Unlike our study at the revision level, they use monthly snapshots of four open source subject systems. The study does not find evidence that cloned code is risky. There is also no strong evidence that cloned code with more siblings is more defect-prone than cloned code with fewer siblings.

Our study differs from similar studies in the literature in several aspects. We explore a new set of predictors related to clone siblings, which were not previously investigated by other studies. When analyzing software systems, we do so at the method revision level. To the best of our knowledge, our study is the first to perform clone-defect analysis at this level. We also use Cox hazard models with stratification. The stratification allows us to model cloned and non-cloned method revisions using different baseline hazards. Hence, we are able to infer the most crucial predictors in determining the risk of a method experiencing a defect fix, to analyze how cloned methods are affected by their siblings and to figure out when resources should be allocated to testing and defect fixing in the lifetime of a method.

VII. CONCLUSION

In this study, we use Cox hazard models to determine whether cloned code is harmful, what features of cloned code make it defect-prone, and when in the lifetime of a method is it most prone to defects. We analyze the models to understand which predictors are significant in determining defects and the relationship between the predictors and the defects. We demonstrate the validity and accuracy of our models using the proportionality assumption test and by calculating the spearman correlation between the actual and predicted occurrence of defects. Based on our study of two systems, we made two findings. First, we found that cloned code is not always more risky than non-cloned code; the risk seems to be system dependent. For example, our study showed that cloned code is more risky than non-cloned code in ArgoUML, unlike in Ant. Second, we discovered that the survival of all methods against defects decreases with time. This indicates that more testing effort should be dedicated to methods with a longer history of commits.

REFERENCES

[1] A. E. Hassan and R. C. Holt, “Studying The Evolution of Software Systems Using Evolutionary Code Extractors,” In Proceedings of the Principles of Software Evolution, 7th international Workshop (IWPSE 2004), IEEE Computer Society, Sept. 2004, pp. 76-81.

[2] A. G. Koru, K. El Emam, D. Zhang, H. Liu, and D. Mathew "Theory of relative defect proneness: Replicated studies on the functional form of the size-defect relationship," Empirical Software Engineering, vol. 13, pp. 473-498, Oct. 2008.

[3] A. Mockus and L. G. Votta. "Identifying reasons for software change using historic databases," In Proceedings of the 16th International Conference on Software Maintenance (ICSM '00), IEEE Computer Society, Oct 2000, pp 120–130.

[4] A. Mockus, P. Zhang, and P. Luo Li, “Predictors of Customer Perceived Software Quality," In Proceedings of the 27th international Conference on Software Engineering (ICSE '05), ACM, May 2005, pp. 225-233.

[5] CCFinder, http://www.ccfinder.net/

[6] C. Kapser and M. W. Godfrey, "'Cloning Considered Harmful' Considered Harmful," In Proceedings of the 13th Working Conference on Reverse Engineering (WCRE '06), IEEE Computer Society, Oct 2006, pp. 19-28.

[7] K. Roy and J. R. Cordy, “A Survey on Software Clone Detection Research,” Queens University, Kingston, ON, Canada, Technical Report No. 2007-541, 2007.

[8] E. Juergens, B. Hummel, F. Deissenboeck, and M. Feilkas “Static Bug Detection Through Analysis of Inconsistent Clones,” In Workshopdand SE Konferenz 2008, LNI. GI, 2008.

[9] F. Rahman, C. Bird, and P. Devanbu. “Clones: What is that Smell?” In Proceedings of the Seventh IEEE Working Conference on Mining Software Repositories (MSR ’10), May 2010, pp.72-81.

[10] J. D. Singer and J. B. Willett, Applied Longitudinal Data Analysis. New York: Oxford University Press, 2003.

[11] J. Fox, "Cox Proportional-Hazards Regression for Survival Data," Appendix to An R and S-PLUS Companion to Applied Regression, February 2002.

[12] J. Harder and N. Göde, "Modeling Clone Evolution," The 3rd International Workshop on Software Clones (IWSC '09), In Proceedings of the 13th European Conference on Software Maintenance and Reengineering, March 2009 pp. 17–21.

[13] L. Aversano, L. Cerulo, and M. Di Penta , “How Clones are Maintained: An Empirical Study,” In Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07), IEEE Computer Society, March 2007, pp. 81-90.

[14] L. Marks. “An empirical study for the impact of maintenance activities on code clones,” M.Sc. thesis, Queen’s University, Kingston, ON, Canada, 2009.

[15] M. Cataldo, A. Mockus, J. A. Roberts and J. D. Herbsleb, “Software Dependencies, Work Dependencies and Their Impact on Failure,” IEEE Transactions on Software Engineering, vol. 35, no. 6, pp. 864-878, November/December 2009.

[16] M. D’Ambros and M. Lanza, “BugCrawler: Visualizing Evolving Software Systems,” In Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR '07), IEEE Computer Society, March 2007, pp. 333-334.

[17] M. Kim and D. Notkin, “Using a Clone Genealogy Extractor for Understanding and Supporting Evolution of Code Clones,” The Second International Workshop on Mining Software Repositories, co-located with International Conference on Software Engineering, pages 1-5, ACM, May 2005.

[18] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan “An Empirical Study on Inconsistent Changes to Code Clones at Release Level,” In Proceedings of the 16th Working Conference on Reverse Engineering (WCRE '09), IEEE Computer Society, Oct 2009, p. 85-94.

[19] R. Koschke, “Survey of Research on Software Clones: Duplication, Redundancy, and Similarity in Software,” Dagstuhl Seminar 06301, 2006.

[20] R, http://cran.r-project.org/

[21] T. Kamiya, S. Kusumoto, and K. Inoue, “CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code,” IEEE Transactions on Software Engineering, vol. 28, no. 7, pp. 654-670, Jul 2002.

[22] T. Therneau, "R Survival Package", http://cran.r-project.org/web/packages/survival/index.html

[23] T. Zimmermann, R. Premraj and A. Zeller, “Predicting Defects for Eclipse,” In Proceedings of the 3rd International Workshop on Predictor Models in Software Engineering (PROMISE’07), IEEE Computer Society, May 2007, pp. 9.