An Empirical Study of Identical Function Clones in CRAN

Maëlick Claes, Tom Mens, Narjisse Tabout, Philippe Grosjean
Software Engineering Lab & Ecologie numérique des Milieux aquatiques Lab
COMPLEXYS Research Institute, University of Mons
Email: firstname.lastname@umons.ac.be

***Abstract*—Code clone analysis is a very active subject of study, and research on inter-project code clones is starting to emerge. In the context of software package repositories specifically, developers are confronted with the choice between depending on code implemented in other packages, or cloning this code in their own package. This article presents an empirical study of identical function clones in the CRAN package archive network, in order to understand the extent of this practice in the R community. Depending on too many packages may hamper maintainability as unexpected conflicts may arise during package updates. Duplicating functions from other packages may reduce maintainability since bug fixes or code changes are not propagated automatically to its clones. We study how the characteristics of cloned functions in CRAN snapshots evolve over time, and classify these clones depending on what has prevented package developers to rely on dependencies instead.**

I. Introduction

Analysing the impact (whether it be harmful or beneficial) of inter-project code cloning is an emerging topic of research in the code cloning community [1], [2]. Developers are often confronted with the difficult choice between depending on existing functions developed in other libraries, or copy-pasting or re-implementing similar functions in their own code. In the case of depending upon a third-party library, errors may be introduced inadvertently when upgrading to a newer version of the library one depends upon. Finding and fixing these errors can be cumbersome. Duplicating functions across different libraries is an alternative but may be detrimental to the maintainability in the long run.

In this article, we study the extent of the practice of cloning function code between packages contained in a large open source package archive maintained by a specific community of developers. In such a complex and dynamically evolving context, there are many packages, maintained by different developers, having many dependencies between them, and being subject to regular changes and updates.

The case study that we have chosen for this paper is CRAN, the official archive network of R packages supported by the community surrounding the statistical project R. The R developer community is part of a rather specific ecosystem that mostly consists of non-programmers such as statisticians, biologists and economists. They generally don't have a software engineering background as strong as other programming language communities.

Identical cloned functions across packages appear to be omnipresent in CRAN. With our study, we aim to understand why this is the case, and how this practice evolves over time. In particular, we want to understand to which extent functions are Type-1 clones across packages, why R package developers clone functions, and if clones could be avoided by the introduction of explicit dependencies. Thus our longitudinal empirical study of inter-package function clones in CRAN focuses on the following research questions:

1) How prevalent are clones in CRAN, and how does this evolve over time?
2) How and why did clones appear?
3) Is it possible to remove clones and how?

II. About CRAN

CRAN is the official R package repository containing thousands of software packages maintained by a community of thousands of developers surrounding the R project. CRAN dates back from 1997, and is still the biggest R package repository available today, containing over 6,000 packages and 9 million lines of R code. While some researchers have studied the evolution of this ecosystem [3], we are not aware of any study that focuses on the presence of function clones in CRAN and the consequences thereof.

CRAN has a rather strict policy to ensure package quality. Packages need to pass a series of tests to be accepted in the repository. These tests are rerun daily on the last version of each available package. Package maintainers must ensure that their package still passes these tests to avoid their package being archived. This policy puts a heavy burden on package maintainers, especially if the package no longer passes the tests due to un update of some dependent package over which the maintainer has no control [4].

We hypothesise that R package maintainers sometimes resort to copy-and-paste reuse to reduce the extent of this problem. Indeed, copying the code of a function they want to reuse from another package requires less effort (at least at the short term) than explicitly depending on that package and take the risk that future changes in that package may lead to conflicts in ones own package.

The other way around, we also suspect that some function clones between R packages exist because the declaration of an explicit package dependency does not allow to access the required function. This is for example the case if the function is local to the package, or if it is anonymous, or if it is a global function that is not exported in the package's namespace.

III. Related work on software clones

Code cloning is an active research topic of the software engineering community. A comprehensive overview of software cloning literature can be found on http://students.cis.uab.edu/tairasr/clones/literature/. Most research focuses on how to detect clones (e.g., [5], [6], [7], [8], [9]), while some articles focus on how to remove clones (e.g., [10], [11], [12]). Research on inter-project software clones is starting to emerge ([1], [2]), and our work fits within this theme. A specificity of our research is that we take an ecosystemic point of view.

Code clones can be classified in four mutually exclusive types [13], [9]. Type-1 clones are syntactically identical code snippets at the abstract syntax level (i.e., ignoring differences in white space, layout and comments). Type-2 clones additionally differ in identifier names and literal values. Type-3 clones syntactically differ by having some statements added, modified and/or removed with respect to each other. Type-4 clones implement the same functionality while being syntactically dissimilar. This article focuses on Type-1 function clones because our goal is to study code that has been duplicated across different packages rather than depended upon. Thus we need to be sure that the identified clones are equivalent from a computational point of view.

Code clones are often considered harmful because they lead to redundancy due to code duplication. This makes software maintenance more difficult. For example, Jürgens et al. found inconsistent changes to code clones to be very frequent and a significant number of defects are introduced by such changes [14]. On the other hand, many situations have been reported in which clones are not considered harmful, are impossible or impractical to remove, or are even beneficial [15], [16], [17].

The bottom-line is that, if you really have to clone some code, you need to do it safely. Proactive tool support can be very beneficial to help detect the presence of clones, to propagate changes across clones, to assess the risk or benefits of clones, and to help remove clones if needed. Many tools have been proposed for detecting clones, including CCFinder, CBCD, CloneDR, CPMiner, Dup, Duploc, iClones, KClone and NiCad. For a qualitative comparison of clone detection techniques and tools, see [9].

In earlier work we presented *maintaineR* [18], a web-based dashboard that can be used by CRAN package maintainers. This tool enables, among others, to see which functions in their package are duplicated in other packages. The current article uses such information to perform an in-depth analysis of identical function clones in CRAN.

IV. Terminology

Before answering the research questions introduced in Section I, let us introduce the necessary terminology and notation.

Let *Date* be the set of all possible dates (represented by year, month and day).[1] This set is totally ordered, i.e., any two dates can be compared using the $<$ operator.

[1]We exclude more specific information such as the hour, minutes and seconds; and we assume that all dates are converted to the same timezone.

Each CRAN package is identified uniquely by its package name. The list of all available CRAN packages, sorted by name, can be found on http://cran.r-project.org/web/packages/available_packages_by_name.html. Packages are versioned, and the code contained in each version may be different. As an example, package biotools has 3 different versions (1.0, 1.1, and 1.2), with different release dates (2014-02-12, 2014-02-16, and 2014-05-26).

Notation IV.1 (Package and version)**.**

P denotes the set of all packages.

$p \in P$ denotes an individual package.

$\forall p \in P : V_p$ denotes the set of all versions of package p.

$succ : V_p \to V_p$ is the (partial) successor function defining a total order on V_p. It is defined for each version of a package except the last one, which does not have any successor.

$date : V_p \to Date$ gives the release date of each package version. By definition, $\forall v \in V_p : date(succ(v)) > date(v)$.

$V = \cup_{p \in P} V_p$ denotes the set of all versions of all packages.

$package : V \to P : v \to p$ if $v \in V_p$

Each package version $v \in V$ is characterised by a number of *function* definitions, that contain source code.

Notation IV.2 (Function)**.** *Let $v \in V$.*

F_v is the set of all functions belonging to a package version v. A function $f \in F_v$ is a triple $(f_{args}, f_{body}, f_{env})$ where f_{args} denotes the set of function parameters, f_{body} the function body, and f_{env} the environment of the function.

$F_v = F_v^G \cup F_v^L$ partitions all functions of F_v into local *functions F_v^L (that are defined internally to another function) and* global *functions F_v^G.*

$contains \subseteq F_v^L \times F_v$ determines inside which other function of the package version a local function is defined.

$F_v = F_v^N \cup F_v^A$ partitions all functions of F_v into named *functions F_v^N (that have an explicit name) and anonymous functions F_v^A (that do not have an associated name).*

$name : F_v^N \to \Sigma^*$ provides the name of each named function as a sequence of symbols belonging to some alphabet Σ.

A *snapshot* S^d represents the set of the latest available versions of each package at a given date d. These snapshots allow us to study the evolution of CRAN over time. The date d of each snapshot has the format YYYY-MM-DD. The latest package versions existing at that date will be considered in the snapshot.

Notation IV.3 (Snapshot)**.** *Let $d \in Date$.*

$S^d = \{w \in V \mid date(w) \leq d \wedge \nexists v \in V_{package(w)} : date(w) < date(v) \leq d\}$.

$F^d = \{f \in F_v \mid v \in S^d\}$ is the set of all functions in snapshot S^d. $F_v^d = F^d \cap F_v$ is the set of all functions in package version v belonging to snapshot S^d.

From this definition, and the fact that all package versions are related by a total order *succ* based on their release date, it

follows that *each snapshot contains at most a single version of each package*. Because of this, whenever we restrict ourself to a particular snapshot, we will use the term *package* to refer to the particular version of this package belonging to the snapshot.

In this article, we are interested in finding identical function clones (i.e., Type I clones) belonging to the same snapshot. We will use the term *clones* to refer to distinct functions with an identical function body (the name, arguments and environment of the functions are allowed to be different). Clones may appear either within the same package, within two versions of the same package, or belong to two different packages. *Clone sets* represent groups of identical clones.

Notation IV.4 (Clones and clone sets)**.** *Let $d \in Date$.*

$C^d = \{(f,g) \in F^d \times F^d \mid f \neq g \wedge f_{body} = g_{body}\}$ *denotes the clone relation between functions. Let $f \in F^d$, then we define $clones^d(f) = \{g \in F^d \mid (f,g) \in C^d\}$.*

C^d forms a partial equivalence relation (i.e., it is symmetric and transitive). A clone set *$C \subseteq C^d$ is a class of (function) clones defined by this partial equivalence relation C_d.*

For each clone set *C we define its* origin(s) *as the function(s) representing the oldest incarnation(s) of the clone:*

$origin(C) = \{f \in C \mid \nexists g \in C : f \neq g \wedge date(g) < date(f)\}$

While theoretically a clone set can have multiple origins, in practice this occurs very rarely. And even if it does, these multiple origins tend to belong to the same package version (i.e., the multiple origins are internal clones of one another in the same package). For example, for snapshot S^d at date d =2014-12-01 we found exactly 1 origin package for all 3,184 detected clone sets.

Our research focuses on inter-package clones, i.e., clones across *different* packages, as they are subject to the maintenance problem described in Section I. We will not consider clones between different versions of the *same* package.

Notation IV.5 (Inter-package function clones)**.**

$C^d_{Inter} = \{(f,g) \in C_d \mid \exists v,w \in S^d : v \neq w, f \in F_v, g \in F_w\}$ *denotes all clones between functions belonging to different packages. Like C^d it forms a partial equivalent relation that allows us to define clone sets.*

$clones^d_{Inter}(f) = \{g \in F^d \mid (f,g) \in C^d_{Inter}\}$.

For example, package biotools 1.2 has one identical clone with package soilphysics 1.1. The function body is

```
if (is.null(text))
 text<-"Welcome to the statistical software revolution!"
if ( !inherits(text,"character") || length(text) != 1)
 stop("'text' must be a character vector of length 1!")
vec <- strsplit(text, "")[[1]]
lab <- c(vec, "\n")
for (i in 1:length(lab)) {
 setTxtProgressBar(txtProgressBar(char = lab[i]), 0.01)
 Sys.sleep(0.1)
 }
```

R packages can be related by dependency relationships specified in the R *DESCRIPTION* file of the package version. There exists different kind of dependencies for R packages: *depends*, *imports*, *suggests* and *enhances*. We limit ourselves in this study to the *depends* and *imports* dependencies as they are the ones that are mandatory to install the package. We only consider dependencies to CRAN packages, and exclude all dependencies to R "core" packages.

Notation IV.6 (Version dependencies)**.** *Let $d \in Date$.*

$Dep^d \subseteq \{(v,w) \in S^d \times S^d \mid v \neq w\}$ *denotes the dependency relation between packages belonging to snapshot S_d.*

$dep^d(v) = \{w \in S^d \mid (v,w) \in Dep^d\}$ *is the set of all direct dependencies of package v.*

$revdep^d(v) = \{w \in S^d \mid (w,v) \in Dep^d\}$ *is the set of all direct reverse dependencies of package v.*

$dep^d_*(v) = dep^d(v) \cup \bigcup_{w \in dep(v)} dep^d_*(w)$ *is the set of all recursive dependencies of v.*

$revdep^d_*(v)$ *is defined in a similar way.*

For example, let us consider d =2012-04-02. dep^d(pgfSweave) contains packages *cacheSweave*, *tikzDevice*, *highlight* and *formatR*. dep^d_*(pgfSweave) contains these packages and their recursive dependencies, which are *slashR*, *parser*, *filehash*, *Rcpp*, *digest* and *codetools*.

V. Type-1 function clone extraction

To analyse the history of clones in CRAN, we have proceeded as follows to extract and identify Type-1 function clones. First, we parsed the source code of each version of each CRAN package using built-in R functions. More specifically, we used the R base function parse to construct the abstract syntax tree (AST) of the body of each function of each package. Working with the AST allows us to ignore all code comments and differences in code indentation between otherwise identical function bodies. Next, we computed a hash value for each function body using the SHA-1 cryptographic secure hash algorithm. Two functions that have the same hash value for their function body can be considered as identical functions with a negligible probability ($< 10^{-18}$) of false positives.

For the purpose of the empirical analysis we excluded all functions whose body contains less than 6 lines of code. Through manual inspection we found that this value allows us to avoid most of the small code fragments leading to "accidental clones".

We also excluded all intra-package clones, i.e., clones that appear within the same package. For the empirical analysis, only those clones that appear between different packages (i.e., belonging to the clone relation C^d_{Inter} of Section IV) are of interest to us.

VI. Observed Clone Cases

Before delving into an empirical analysis, we focused on a limited subset of "interesting" CRAN packages with respect to their cloning behaviour. In particular, we considered those packages for which there is an unusually high number of clones, or an unusually high number of packages that have cloned functions belonging to the considered package. The aim of this section is not to provide a representative classification

but is rather indicative about some interesting cases of clones found in CRAN. We present these observed cases of cloning behaviour below.

1) Coexisting package versions: In some CRAN snapshots, two different "versions" of the same package may coexist. While these packages have a different name, one of them can be regarded as the new version of the other. Needless to say, the majority of functions from the old package will be cloned in the new package. A valid reason for this clone case is to allow existing packages to continue to depend on the old version, while already exposing the new version with extended functionality.

One occurrence of this type of clone behaviour was found for packages *plyr* and *dplyr*. They are maintained by the same person, and both allow to manipulate R data structures more easily and more efficiently but in a different way. They share 3 identical function clones totalling 48 lines of code.

Another occurrence are packages *lme* and *nlme* that have coexisted for some time. Both packages fit the same goal of providing statistical model functions. *nlme* adds non-linear models to *lme* and actually replaced it in later snapshots of CRAN. They share more than 600 identical function clones totalling over 7,000 lines of code.

A third example is the pair of packages *np* and *npRmpi*. The latter package is a version of *np* that uses MPI (Message Passing Interface) to distribute computation. Both are currently available on CRAN, are maintained by the same person and share more than 10,000 lines of code.

2) The fork *package:* Related to the previous case, *forked* packages continue to coexist with the package they have forked from. An example is package *Rcmdr*, offering a graphical interface to use R statistical functions. Package *QCAGUI* provides a graphical interface for the *QCA* package, and can be considered as a fork of *Rcmdr* with most of the statistical related features removed. *Rcmdr* and *QCAGUI* share more than 8000 lines of code.

3) The frequently cloned package: For some packages, most functions have been cloned by other packages. An example is *distr*, which contains 182 lines of code, and all its global functions have been cloned by different packages.

4) The utility package: We refer to an utility package as a package that bundles together a lot of functions that are cloned from many other packages. An example is package *DescTools*, which gathers functions for basic statistics that are scattered across different packages, and bundles them together into a single package. *DescTools* copied 52 functions (totalling 1,419 lines of code) from 27 different packages. Some of them are public functions while others are local functions meaning that they are probably used in wrapper functions.

5) The popular package: A popular package contains specific functions that are cloned by a lot of other packages. An example of such a package is *MASS*, a well-known and widely reused statistical package. It has 16 functions that have been cloned by 16 different packages for a total of 180 code lines.

6) The popular function: A popular function is a function that is cloned by a lot of different packages, while the other functions of the same package are not. An example is the package *combinat*, whose function *permn* of 151 lines of code is cloned by 7 different packages.

VII. Metrics

In this section, we define the metrics that we will use for the empirical analysis in Section VIII.

A. Size metrics

Let $d \in Date$, S_d the corresponding snapshot, $v \in S_d$ a package (version), and $f \in v$ a function.

$LoC(f)$ = number of lines of code of f.

$LoC(v) = \sum_{f \in v} LoC(f)$ = number of lines of code of v.

We define size metrics at the snapshot level as follows:

$NoP(d) = \mid S^d \mid$ = number of packages in snapshot S^d.

$LoC(d) = \sum_{v \in S^d} LoC(v)$ = lines of code for snapshot S^d.

B. Clone metrics

$NoC(f) = \mid clones^d_{Inter}(f) \mid$ = number of inter-package Type-1 clones of function f in the same snapshot S_d.

Let $Clones(v) = \{f \in v : NoC(f) > 0\}$ the set of function clones contained in package version v.

$NoCF(v) = \mid Clones(v) \mid$ = number of cloned functions of package v.

$LoCC(v) = \sum_{f \in Clones(v)} LoC(f)$ = number of lines of cloned code in package v.

$RoCS(v) = \frac{LoCC(v)}{LoC(v)}$ = ratio of cloned code in package v.

We can aggregate these clone metrics at the snapshot level as follows.

$NoCP(d) = \mid \{v \in S^d : NoCF(v) > 0\} \mid$ = number of packages containing clones in snapshot.

$NoCS(d)$ = number of clone sets of snapshot d = number of classes defined by the partial equivalence relation C^d_{Inter}.

$LoCC(d) = \sum_{v \in S^d:\ NoCF(v) > 0} LoCC(v)$ = number of lines of cloned code in snapshot.

$LoCCP(d) = \sum_{v \in S^d:\ NoCF(v) > 0} LoC(v)$ = number of lines of code in all packages containing clones.

$RoCP(d) = \frac{NoCP(d)}{NoP(d)}$ = ratio of packages with clones.

$RoCC(d) = \frac{LoCC(d)}{LoC(d)}$ = ratio of cloned lines of code.

$RoCCP(d) = \frac{LoCCP(d)}{LoC(d)}$ = ratio of cloned lines of code in packages containing code.

All these metrics can be qualified by an extra parameter n representing a minimal threshold on the function size expressed in lines of code, i.e., we restrict the functions under consideration to $\{f \in F^d \mid LoC(f) > n\}$. In this paper we have used a threshold $n = 5$ to exclude all clones containing less than 6 lines of code.

VIII. Empirical Analysis

This section uses the metrics defined in Section VII to answer the research questions presented in the introduction.

A. How prevalent are clones in CRAN?

In order to assess the importance of the cloning phenomenon across CRAN packages we computed the snapshot for each day d from January 2000 to December 2014. For each snapshot S^d we computed the snapshot-level metrics related to clones.

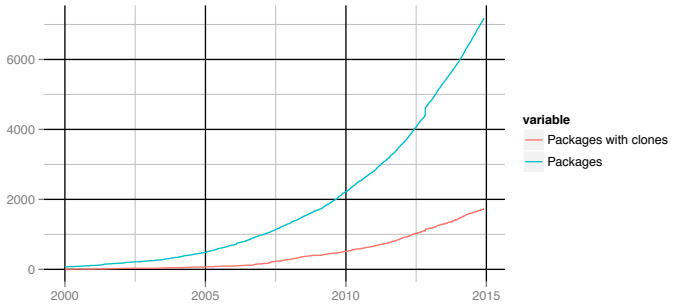

Fig. 1. Evolution over time of $NoP(d)$ (in blue) and $NoCP(d)$ (in red).

Figure 1 shows the evolution in CRAN of the number of packages $NoP(d)$ and number of packages containing clones $NoCP(d)$. The general trend is that the number of packages containing clones increases over time, up to 2,000 packages containing clones today. This corresponds to 24.2% of all packages. The trend follows the overall exponential growth trend of the number of available CRAN packages.

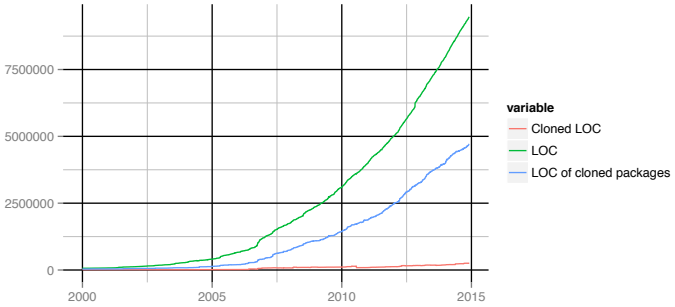

Fig. 2. Evolution over time of $LoC(d)$ (in green), $LoCC(d)$ (in red) and $LoCCP(d)$ (in blue) in CRAN.

The evolution of the number of lines of cloned code $LoCC(d)$ (Figure 2) also follows an increasing trend. We observe that $LoCC(d)$ is much smaller than $LoCCP(d)$, the total number of lines of code of the packages containing these clones. The ratio amounts to 2.6% of all lines of code in CRAN and 5.3% of all lines of code of packages containing clones. This is much less than the ratio observed in Figure 1 of 24.2% of packages containing clones. Nevertheless, these cloned functions are included in packages that, together, represent 49,7% of all lines of code in CRAN!

Figure 3 shows how these ratios evolve over time. The ratio $RoCC(d)$ of lines of cloned code decreases over time (starting from around 20% initially to less than 5% today). The ratio $RoCP(d)$ of packages containing cloned code has the opposite

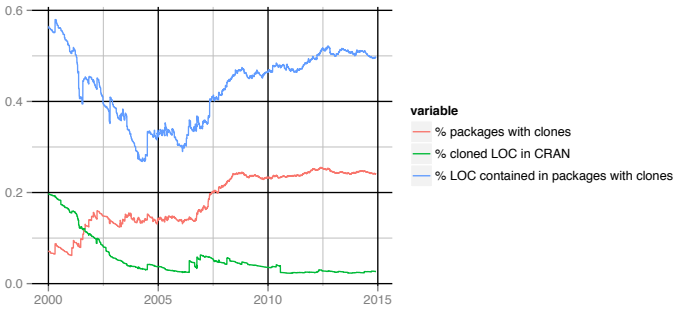

Fig. 3. Evolution over time of $RoCP(d)$ (in red, the ratio of packages containing clones), $RoCC(d)$ (in green, the ratio of cloned lines of code) and $RoCCP(d)$ (in blue, the ratio of lines of code in those packages containing cloned code).

behaviour, with a higher percentage of packages is containing cloned code. In both cases, the ratio seems to stabilise during the last 8 years of CRAN. We hypothesise that this is because CRAN has become more mature.

From these findings we can conclude that

- The cloning phenomenon in CRAN impacts quite a lot of packages (up to 25%). However it does not impact the majority of CRAN packages.
- The ratio of packages impacted by cloning appears to have stabilised.
- While cloning impacts very few lines of code compared to the overall size of CRAN, it still impacts more than 250,000 code lines. Moreover, those lines are included in packages that represent around 50% of all code lines in CRAN.

B. Why did clones appear?

We have seen previously that cloning potentially impacts hundreds of thousands of lines of code. Our goal is to understand the reason of existence for those clones and whether these clones could have been avoided.

To fulfil this goal we study in more detail snapshot S^d corresponding to date d =2014-12-01. We limit ourselves here to those packages that are not archived at date d.[2]

We counted $NoP(d) = 6,253$ non-archived packages. The clone relation C^d_{Inter} resulted in $NoCS(d) = 3,184$ clone sets, involving 7,366 function clones in $NoCP(d) = 1,409$ distinct packages. In total, this amounts to $LoCC(d) = 162,327$ lines of cloned code out of $LoC(d) = 8,338,417$ in total (i.e., $< 2\%$).

In order to understand why these clones exist we studied $origin(C)$, the origin function of each identified clone set C. The origin corresponds to the function with the oldest date, implicitly assuming that all other clones belonging to the same clone set were copied from it. We found exactly 1 origin package for all 3,184 considered clone sets.

For the origin of each clone set, we try to answer the following questions:

[2]In the previous research question we did not exclude archived packages because there is no history available online to know which packages were archived or not at a certain point in time.

- Is the origin *anonymous* (i.e., not stored in any variable)?
- Is the origin declared *locally* (i.e., declared inside another function)?
- Is the origin available as a *public* function to the package users?
- Does the origin still exist in the most recently available package version?

Among the 3,184 considered origin functions (one for each clone set), we identified 796 functions (i.e., 25%) that were either anonymous or local. 250 of these were both anonymous and local.

For the 2,388 (3,184 - 796) origin functions that were globally visible (i.e., not local) in the origin package version, 202 (8.45% of all global origin functions) were no longer available in the latest considered version of the same package, either because the function has been removed or changed somehow over time.

The current CRAN policy requires packages to define a "NAMESPACE" file that lists which functions or objects are exported by the package. Those exported functions are all the functions that the package user is allowed to use[3]. Because NAMESPACE files can use regular expressions and because package environments can be modified dynamically, we extracted the list of exported objects by loading each package in a virtual machine containing a snapshot of CRAN corresponding to the release date of the package.

Out of the 2,186 (2,388 - 202) origin functions that still exist today, we weren't able to retrieve the list of exported functions for 287 of them (i.e., 13%). Of the remaining 1,899 origin functions, 673 were exported while 1,226 were not.

In summary, it turns out that, for the considered snapshot, cloning cannot be avoided for the majority of clones in each identified clone set. Of the 3,184 origin clones, 25% (796) of them were local functions that cannot be depended upon no matter whether they are exported or not by their containing package. Of the remaining 1,899 global functions, only 35% (673) were public ones.

Fig. 4 presents the distribution of number of lines of code $LoC(f)$ for each clone set origin function f. The function size varies a lot, and while most origin functions tend to be rather small (less than 20 LoC for more than 50% of them), their size can increase up to 1,000 LoC. We also observe that function size tends to be bigger for global than for local origins, and bigger for public clones than for private origins.

C. Is it possible to remove clones and how?

a) Removing clones by adding a dependency to the origin: We have seen that it was only possible for a small fraction of the clone sets to remove identical clones by adding a dependency to the origin function. However, we still need to check whether this dependency already exists or if it could be added. This dependency cannot be added if the package containing the origin (directly or indirectly) depends on the package containing the clone, since otherwise a cyclic dependency would be introduced.

[3]Although, technically, it is still possible to call non-exported functions using syntactic sugar, this is strongly discouraged by the CRAN check process.

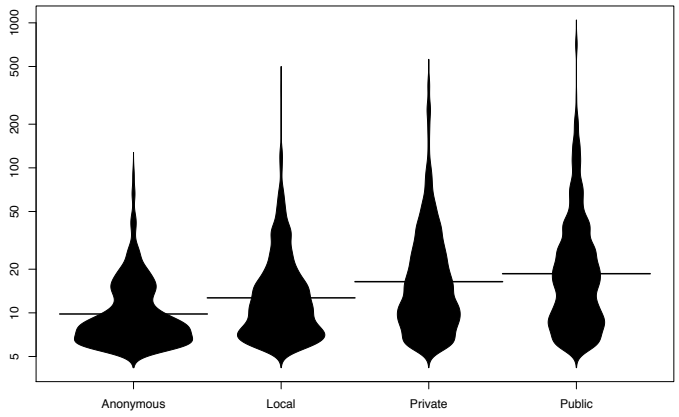

Fig. 4. Beanplots showing the distribution of $LoC(f)$ for the clone set origins, classified according to their visibility: anonymous, local, (global) private or (global) public.

To the previously identified 673 public (and hence, potentially refactorable) origin functions correspond 782 clones in 332 packages. Of these, there are 49 packages with an existing direct dependency to the origin package. In the opposite direction we found 20 packages for which their origin package directly depends upon them and only one for which the origin package indirectly depends upon it.

b) Removing clones by adding a dependency to a clone of the origin function: While the majority of clones cannot be removed by depending upon the package that contains the origin function, perhaps a dependency can be added to another package containing a clone of the origin.

Let us consider the 3,458 non-origin clones of the 2,511 clone sets for which the origin is not refactorable by adding a dependency. Is one of the non-origin clones refactorable instead? For 801 clone sets, all non-origin clones are all local functions, and for 194 clone sets we were not able to retrieve the list of exported objects by the package. For the remaining global functions for which we could retrieve exported objects, 1,266 are private functions and only 250 are public ones.

For the 250 clone sets containing at least one public clone, there is a total of 317 clones that could potentially be removed by adding a dependency to the package with this public clone. 31 already have this dependency and 18 have a reverse dependency to the package declaring the public clone, making it impossible to add the dependency.

IX. Threats to Validity

Our study has several potential threats to validity. Since we have restricted ourselves to R packages, our results do not necessarily generalise to other package-based systems.

For our analysis we used mainly tools and scripts that we developed ourselves and which could still contain bugs. We also relied on data available on CRAN web site and some of this information could be unreliable. In particular we cannot be sure that the release date of packages is the actual one as it can be misestimated by a few days.

For snapshots older than one year there is no way of knowing which packages were present on CRAN at that time. The history of when package versions were archived is not available and we have to rely on data we started to extract since September 2013.

The threshold of at least 6 code lines we have used to consider identical functions to be clones is arbitrary and could lead to over- or underestimations of the number of clones.

X. Future Work

This paper studied the presence of Type-1 function clones across CRAN packages in an objective way. We intend to complement this information by performing a subjective survey with actual R package maintainers. In particular, we wish to know to which extent they purposefully resort to the practice of code cloning, and if they perceive the presence of clones as something good or bad. A survey also allows us to get direct feedback from CRAN package developers.

So far we only considered Type-1 clones. A direct extension would be to consider Type-2 and Type-3 clones as well. For example, it would be interesting to find out how long it takes before a Type-1 clone becomes a Type-2 clone, and so on. Further work is needed to understand how function clones in CRAN packages evolve over time in order to help package maintainers cope with these clones in a better way.

Section VI highlighted a series of interesting clone cases. Future work includes a systematical study of clone patterns in order to determine if those cases are representative of cloning behavior in CRAN.

Finally, we intend to explore how popularity and specialisation of CRAN packages impacts presence of function clones. An answer to this question could both highlight why the origins of function clones tend to be concentrated in a small number of packages and why a lot of packages do not encounter cloning.

XI. Conclusion

This article studied the problem of inter-project software clones from an ecosystemic point of view. To this extent, we carried out an empirical study of Type-1 function clones across R packages contained in the CRAN package repository over a 15-year time period. Our goal was to understand to which extent functions are cloned across packages, why R package maintainers clone functions, and if clones could be avoided.

While identical cloned functions of at least 6 code lines appear to be present in a rather small portion of the code of all packages, they still represent hundred of thousands of lines of code. Moreover, they are present in one out of four packages that together make up half of all CRAN code.

We were able to identify an important number of clones that could theoretically have been avoided by introducing explicit dependencies to another package containing the function clone. Only in very few cases it was not possible to add such a dependency because it would give rise to a cyclic dependency.

We also found valid reasons why cloned functions appeared. In the majority of cases, cloning could not be avoided because the original function being cloned was local or private. This made it technically impossible to reuse the function by simply depending upon the package defining it.

Hence, the problem of identical cloned functions in CRAN appears to be less problematic than what one could expect at first. Nevertheless, we believe that R package maintainers still lack information about, and could benefit from, feedback on the presence of clones in their package and dedicated tools to help them deal with it.

Acknowledgements

This research was carried out in the context of ARC research project AUWB-12/17-UMONS- 3.

References

[1] R. Koschke, "Large-scale inter-system clone detection using suffix trees and hashing," *Journal of Software: Evolution and Process*, vol. 26, no. 8, pp. 747–769, 2014.

[2] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *Int'l Conf. Software Maintenance and Evolution*, 2014, pp. 476–480.

[3] D. M. Germán, B. Adams, and A. E. Hassan, "The evolution of the R software ecosystem," in *European Conf. Software Maintenance and Reengineering*, 2013, pp. 243–252.

[4] M. Claes, T. Mens, and P. Grosjean, "On the maintainability of CRAN packages," in *Int'l Conf. Software Maintenance, Reengineering, and Reverse Engineering*, 2014, pp. 308–312.

[5] K. Kontogiannis, R. D. Mori, E. Merlo, M. Galler, and M. Bernstein, "Pattern matching for clone and concept detection," *J. Automated Software Engineering*, vol. 3, no. 1/2, pp. 79–108, June 1996.

[6] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Int'l Symp. Static Analysis*, Jul. 2001, pp. 40–56.

[7] F. Lanubile and T. Mallardo, "Finding function clones in web applications," in *European Conf. Software Maintenance and Reengineering*, 2003, pp. 379–386.

[8] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," in *Working Conf. Reverse Engineering*, 2006, pp. 253–262.

[9] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

[10] R. Komondoor and S. Horwitz, "Eliminating duplication in source code via procedure extraction," UW-Madison Dept. of Computer Sciences, Technical Report 1461, Dec. 2002.

[11] Y. Higo, S. Kusumoto, and K. Inoue, "A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system," *J. Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 6, pp. 435–461, 2008.

[12] R. Koschke, *Software Evolution*. Springer, 2008, ch. Identifying and Removing Software Clones, pp. 15–36.

[13] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. M. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Trans. Soft. Eng.*, pp. 577–591, Sep. 2007.

[14] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Int'l Conf. Software Engineering*, 2009, pp. 485–495.

[15] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy, "An empirical study of code clone genealogies," in *ESEC/FSE*. ACM, 2005, pp. 187–196.

[16] C. Kapser and M. W. Godfrey, "'Cloning considered harmful' considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, no. 6, pp. 645–692, 2008.

[17] R. K. Saha, M. Asaduzzaman, M. F. Zibran, C. K. Roy, and K. A. Schneider, "Evaluating code clone genealogies at release level: An empirical study," in *Int'l Working Conf. Source Code Analysis and Manipulation*, 2010, pp. 87–96.

[18] M. Claes, T. Mens, and P. Grosjean, "maintaineR: A web-based dashboard for maintainers of CRAN packages," in *Int'l Conf. Software Maintenance and Evolution*, 2014, pp. 597–600.