An Empirical Study of API Breaking Changes in Bioconductor

Hemayet Ahmed Chowdhury

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science And Application

Na Meng, Chair

Song Li

Muhammad Ali Gulzar

November 30, 2022

Blacksburg, Virginia

Keywords: Bioconductor, R, Backward-Incompatibility, API, Breaking Changes

Copyright 2023, Hemayet Ahmed Chowdhury

An Empirical Study of API Breaking Changes in Bioconductor

Hemayet Ahmed Chowdhury

(ABSTRACT)

Bioconductor is the second largest R software package repository that is primarily used for the analysis of genomic and biological data. With downloads exceeding millions in recent years, the widespread growth of the repository's adoption can be attributed to it's diverse selection of community-created packages, written in the programming language R, that allow statistical methodologies for analysis and modelling of data. However, as these packages evolve, their APIs go through changes that can break existing user code. Fixing these API breaking changes whenever a package is updated can be frustrating and time-consuming, especially since a large fraction of the user community are researchers who do not necessarily have software engineering background. In that context, we first present a tool that can detect syntactic API breaking changes between two released versions of a library written in R through static analysis of the package source code. This tool can be of utility to R package developers, so that they can more comprehensively report or handle the breaking changes in their releases, and to R package users, who want to be aware of the API differences that may exist between two releases before upgrading the libraries in their code. Through the use of this tool and manual inspection, we also conducted an empirical study of the breaking changes and backward incompatibility in Bioconductor packages. We studied the 100 most downloaded packages in the repository and found that 28% of all packages releases are backward incompatible. We also found that 55% of these breaking changes go undocumented and developers don't maintain semantic versioning for 22% of the releases. Finally, we

manually inspected 10 library releases that consisted of breaking changes and found 2% of the API-s to affect 31 client projects.

An Empirical Study of API Breaking Changes in Bioconductor

Hemayet Ahmed Chowdhury

(GENERAL AUDIENCE ABSTRACT)

Bioconductor is a software repository that consists of over 2000 software libraries. These libraries can provide users with reusable functions, or APIs, to perform statistical and graphical data analysis. The developers of these libraries will generally make timely updates to the library source code and the functions for various maintainability purposes. However, when clients install these library updates in their existing code, their code might not compile, run or behave the same way it used to anymore due to the changes made in the APIs of the libraries. Such a library release that consists of changes that can potentially break older code is considered to be backward incompatible. Without proper documentation from the library developer's side, fixing these issues can be time consuming as the client might have to manually look at the changes made in the library's source code. In order to tackle this issue, we first present a tool that can analyse two versions of a library and identify a subset of the breaking changes in the API. This can be helpful for both the users and the developers of the libraries to be aware of any breaking changes that exist in a new release. Afterwards, we conduct a study on the Bioconductor ecosystem to see how serious the problem of backward incompatibility really is by studying the top 100 most downloaded packages from the repository. We see that 28% of the releases across these 100 packages are backward incompatible. Since clients are likely to be using multiple libraries at once, this figure can potentially cause frequent issues in client code. We then go on to check how often developers maintain the correct release protocols when updating their libraries. These include versioning the releases

in correct ways, so as to let the users be aware of what releases may be backward incompatible and documenting any breaking changes that occur in a NEWS file that users have access to. In that aspect, we find that 22% of the releases are not versioned correctly and roughly 55% of the breaking changes in the API are not documented. Finally, we investigate how frequently these breaking changes can actually affect client code. Here, we manually inspect 10 releases with a high number of a subset of the breaking changes and find 31 projects that implement these APIs, which would break upon a library update.

Acknowledgments

I would like to acknowledge and thank Professor Na Meng, Professor Song Li and Professor Muhammad Ali Gulzar for agreeing to serve on my committee. Prof. Meng has been a great advisor for me throughout my time at Virginia Tech and I will always be grateful to her for her patience and guidance as a mentor during my first steps in academic research. Prof. Song provided the primary motivation for this research and I'd like to thank him for his support especially during the earlier phases of this project. The course I had taken under Prof. Gulzar in my first semester allowed me to explore some methodologies and techniques that were very relevant for both this research and my potential future career as a software engineer, for which I am very thankful.

Contents

List of Figures **x**

List of Tables **xii**

1 Introduction **1**

2 Background **5**

2.1 Bioconductor 5

2.2 The R Programming Language 5

2.3 R Packages 6

2.4 Semantic Versioning 6

2.5 R Functions 7

2.6 Parameters in R Functions 9

3 Related Work **10**

3.1 API Evolution and Stability 10

3.1.1 API Evolution 10

3.1.2 API Deprecation 11

3.2 The R Ecosystem 12

3.3 Client Migration Support Tools 13

3.4 Breaking Change Detection Tools 14

4 Methodology **15**

4.1 Data Acquisition 15

4.2 A Breaking Change Detection Tool for R libraries 16

4.2.1 Scope of the Tool 16

4.2.2 Tool Overview 17

4.2.3 Parsing the R source code 18

4.2.4 Function Extraction 20

4.2.5 Public API Filtration 22

4.2.6 Comparing Public API Models And Reporting Breaking Changes . . 23

5 Results **24**

5.1 RQ1 : How frequently are package releases backward incompatible? 24

5.2 RQ2 : What is the distribution of the different types of breaking changes across the packages? 26

5.3 RQ3 : How often do developers have semantic versioning issues when releasing a new version? 28

5.4 RQ4 : To what extent do developers document breaking changes? 30

5.5 RQ5 : How frequently do broken APIs occur in client projects? 32

6 Discussion **34**

7 Threats To Validity **37**

8 Future Work **38**

9 Conclusions **40**

Bibliography **41**

List of Figures

2.1 Directory structure of a Bioconductor package 7

2.2 S3 Function Implementation 8

2.3 S4 Function Implementation 8

2.4 S3 and S4 function simultaneous implementation 9

2.5 A setReplaceMethod function implementation in R 9

2.6 Parameters in R Functions 9

4.1 Overview of the API breaking change tool 17

4.2 Sample Function Declaration in R 19

4.3 XML tree representation of a function 19

4.4 S3 and S4 function dispatch methods 21

4.5 S3 and S4 function dispatch methods 21

4.6 S4 function declared using an S3 function 22

4.7 Public function declarations in the NAMESPACE file 23

5.1 Number of packages against percentage release versions that consist of a breaking change 24

5.2 Number of (a) all breaking changes (b) function removals (c) non-default parameter modifications (d) minor parameter modifications that occur across all releases of a package 27

5.3 Number of (a) all breaking changes (b) function removals (c) non-default parameter modifications (d) minor parameter modifications for every package normalised by the number of releases 27

5.4 Distribution of packages with semantic versioning issues in their percentage releases 28

5.5 Distribution of packages with semantic versioning issues in their percentage releases that consisted of a breaking change 29

5.6 Percentage of undocumented breaking changes of different types across the 91 packages 31

List of Tables

5.1 Releases with Semantic Versioning Issues 25

5.2 Package with no breaking changes and their version numbers 25

5.3 Types of breaking changes and their count across all packages 26

5.4 Types of undocumented breaking changes 30

5.5 Breaking changes and their impact on client projects 32

Chapter 1

Introduction

It is only natural that libraries and their APIs will evolve over time, just like every other aspect of the software development process. However, at times the newer releases of these libraries become backwards incompatible. This can become problematic for user productivity if it occurs at a high frequency across a large number of libraries in a software pacakge repository since a lot of time and effort goes into fixing existing code to adapt to the API changes once the libraries are updated. Bioconductor is one such package repository which seems to be particularly susceptible to the issue of breaking changes, as evident from a flurry of stack overflow[9][10][8] and blog posts[2][3] that have been around for quite a while. API breaking changes being an issue in this ecosystem is also fairly intuitive, since Bioconductor comes out with 2 annual releases of all it's packages, while the language the libraries are written in, R, operates on it's own mechanism of 1 release per year. This complex nature of frequent whole-sale releases in the entire ecosystem, added to the fact that most users of Bioconductor are researchers from genomic or biomedical engineering field and do not particularly have a strong background in software engineering means that API breaking changes in the ecosystem do have the potential of becoming a very frustrating and time consuming problem for it's users.

The entire R programming domain itself has seen explosive growth over the recent years as the need for statistical analysis, modelling and prototyping grew across all research domains in general. In that context, quite a few studies have been done in order to understand the

development and growth of R packages[20], maintainability of R package ecosystems[17] and the inter dependency issues between the R ecosystems[21]. However, while some of these studies laterally touched the issue of breaking changes in APIs and briefly talked about the problem, such as the one by Bogart et al.[14] where they investigated why developers create breaking changes in R packages through interviews, we did not find any studies that directly addressed the problem from a quantitative perspective in depth for R repositories. Additionally, we see that most of these studies were focused on CRAN while Bioconductor remained a severely understudied ecosystem despite being the second largest R package repository and having existed for over 20 years.

All of the above works as our motivation for this research. First, despite the growing interest both in terms of research and application in the area of backwards incompatibility of R APIs, we see that no such tool currently exists online that can analyse breaking changes in R libraries. To bridge that gap we present a tool that can statically analyse R code and package files to detect syntactic API breaking changes between releases. The general idea of the tool is to parse the source code into Abstract Syntax Trees and then crawling the trees to extract the public APIs for each package to compare them. Besides our research utilities, this tool can also be used by R package developers to be aware of the existing breaking changes in their package releases so that they can handle or report them better and package users, to get an understanding of the potential code breaking changes before upgrading a library in their code. We then go on to use this tool and additional manual inspection to perform an empirical study of API breaking changes in Bioconductor to see how serious the issue of backwards incompatibility really is in this repository and if it needs further research attention.

In that context, the goal of this study is three-fold : to quantitatively analyse the problem of breaking changes in the Bioconductor ecosystem, raise developer awareness of the underlying

issues and potentially promote further research in the area of API breaking change detection and automatic API migration support tools for R. In this paper, we investigate 5 research questions in total. RQ1 and RQ2 pertain to understanding the nature of API breaking changes in Bioconductor. With RQ3 and RQ4 we investigate if developers maintain the right development protocols to alleviate the issues. And finally, with RQ5 we try to understand how these breaking changes affect client projects. The research questions are as follows :

- **RQ1: How frequently are package releases backward incompatible?**

  We start by checking how often package releases don't maintain backwards compatibility. We use our tool to compare consecutive versions of releases for all the 100 packages and present the data in a clustered column format to get an understanding of the distribution.

- **RQ2: What is the distribution of the different types of breaking changes across the packages?**

  For this research question, we try to understand the nature of the syntactic breaking changes in terms of their types. We primarily define three broad categories of changes that developers make in the function signatures that can cause client code to break. The goal of the question was to understand what types of breaking changes occur the most frequently and perhaps deserve the most attention.

- **RQ3 : How often do developers have semantic versioning issues when releasing a new version?**

  While evolution and change is a natural part of the software development process, developers should generally maintain best practices to let the users of their packages know how and when they change. One such protocol common in the package management domains is the use of correct semantic versioning for releases. If semantic

versioning is properly utilised for packages, clients can get a prior understanding of whether updating a library will break their existing code or not. For this RQ, we investigate how often developers maintain semantic versioning in their package releases and whether a lack of maintaining protocol should be a cause of concern or not.

- **RQ4: To what extent do developers document breaking changes?**
  This research question also has to do with developer best practices when it comes to alerting clients of the changes made in the packages. Bioconductor, like most other major R repositories, recommends developers to update a NEWS.md file for every release in a package, especially with mention of the breaking changes made in the update in order for clients to get a comprehensive understanding of what types of changes to expect. Here, we use our tool to detect the syntactic breaking changes in a package release and check how often developers actually document them in their NEWS file for the awareness of their clients.

- **RQ5: How frequently do broken APIs occur in client projects?**
  Finally, we try to understand how often these breaking changes actually impact client projects. For this, we turn to github and cross check how frequently the breaking changes in our study appear in real life client projects.

The rest of this paper is structured as follows. In section 2, we provide some exposure to the foundational background necessary to understand the context of the study. In section 3, we discuss some related work that acted as motivation behind this paper and also how our work differs from previous studies. In section 4, we discuss in detail the approach we used to build our tool to detect syntactic breaking changes in R libraries. In section 5, we lay out the results of our experiments along with the answers to the research questions and how we reached them. Finally, in section 6 we discuss the implications of our findings.

Chapter 2

Background

2.1 Bioconductor

Bioconductor[1] is an open-source package repository that is primarily used for the analysis of genomic and biological data[23]. Packages in Bioconductor are primarily written in the statistical programming language R. Every year, Bioconductor packages have 2 releases, which follow the semi-annual release of R. One release version corresponds to the release version of R while the other corresponds to the development version. Most users of Bioconductor packages will find the stable release version sufficient for their needs.

2.2 The R Programming Language

R is a statistical computing and graphics oriented programming language. It is maintained by the R Core Team and the R Foundation for Statistical Computing[25]. Over the years, R has become increasingly popular in its use in statistics, bioinformatics, and data science in general. R is based on an open-source free software environment within the GNU package, provided by the GNU General Public License. Written fundamentally in C, Fortran, and R itself, the language has evolved to support procedural programming with its traditional functions and object-oriented programming through generic functions[45].

2.3 R Packages

The R community and its user-created packages are perhaps one of the biggest driving points behind the language's widespread adoption and use in statistical analysis of data. These packages are easy to install and provide statistical methodologies, graphical interfaces, import-export functionalities and strong documentation[16][44]. The packages provide reusable functionalities that are used by researchers to analyse data and create reproducible research elements.

While base packages are provided with the installation of the language, additional packages are available on software repositories such as CRAN[5] and Bioconductor.

R packages generally adhere to stricter specifications compared to other packages. R's built-in package management enables installation of packages that maintain the guidelines for source code, documentation, package metadata and directory structure privded by The Writing R Extendsions manual[43]. Packages provided on repositories like Bioconductor or CRAN also require additional standards to be met. Figure 2.1 shows an example of the directory structure of a Bioconductor package release. Some relevant information about the directory structure that pertain to our study include the source code files are generally located in the R directory and the NAMESPACE file containing the list of functions that are to be exported as public APIs.

2.4 Semantic Versioning

Software package releases are typically versioned with 3 numbers, xx.yy.zz where xx referes to the MAJOR version, yy refers to the MINOR version and zz refers to PATCH version. Whenever a release makes backward incompatible changes, the MAJOR version number

Name	Date modified	Type	Size
.github	8/29/2022 3:23 PM	File folder	
inst	8/29/2022 3:23 PM	File folder	
man	8/29/2022 3:23 PM	File folder	
R	8/29/2022 3:56 PM	File folder	
tests	8/29/2022 3:23 PM	File folder	
vignettes	8/29/2022 3:23 PM	File folder	
.gitignore	3/14/2022 6:58 PM	Text Document	1 KB
.Rbuildignore	3/14/2022 6:58 PM	RBUILDIGNORE File	1 KB
DESCRIPTION	3/14/2022 6:58 PM	File	2 KB
Dockerfile	3/14/2022 6:58 PM	File	1 KB
NAMESPACE	3/14/2022 6:58 PM	File	7 KB

Figure 2.1: Directory structure of a Bioconductor package

should be incremented. MINOR version number should be incremented if functionality is added to the package in a backwards compatible manner. PATCH version number should be incremented if backward compatible bug fixes or refactorings are made. This manner of versioning is known as semantic versioning.

2.5 R Functions

R currently has two different generations of function implementation systems, the S3 and S4. S3 is the original simpler system of function declaration, while the S4 is more formal, requires more validation and provides methods and classes that standardise object oriented programming in R better. In modern R, developers can simultaneously make use of both the versions.

The example in figure 2.2 shows an example of an S3 function that takes an object as it's parameter and returns an attribute of the object.

```
side <- function(object){
object@side
}
```

Figure 2.2: S3 Function Implementation

The S4 implementation of such a function is shown in figure 2.3. Here, using the parameter of the method setMethod(), it is stated that the function expects an object of type Polygon.

Additionally, S3 and S4 functions can be combined for the implementation of the same function but in different names, as shown in figure 2.4. This is a practice developers often follow to make sure their package has all possible implementations of the correct function.

```
setMethod("side", "Polygon", function(object){

object@side

})
```

Figure 2.3: S4 Function Implementation

Another S4 practice of function implementation in R is through the use of setReplaceMethod(), as shown in figure 2.5. While the functionality of setReplaceMethod() can be achieved through setMethod() itself, it is often considered best practice to use this mode of function implementation to assign value to variables or object attributes. More details about R function implementations can be found in the R-Manuals[42].

```
setClass("Polygon", slots = c(side="numeric"))

getSide <- function(object){
object@side
}

setMethod("side", "Polygon", getSide)
```

Figure 2.4: S3 and S4 function simultaneous implementation

```
setReplaceMethod("side", "Polygon", function(object, value){
object@side<-value
})
```

Figure 2.5: A setReplaceMethod function implementation in R

2.6 Parameters in R Functions

Parameters can be defined in quite a few ways in R function signatures. From figure 2.6, we can see that parameters can be defined without any default values, like a and b, while they can also be declared with a default value, like c, which has a default value of 0. In that case, a and b are two parameters that are required as inputs when calling the function, while an input for parameter c is optional, and it will take the value of 0 if it's not used.

```
addNumbers <- function(a, b, c=0){
a+b+c
}
```

Figure 2.6: Parameters in R Functions

Chapter 3

Related Work

Despite being the second largest R distribution package repository and the primary distribution center for genomic data analysis packages for R, the research in trying to understand the health of the Bioconductor ecosystem has been minimal. Studies related to our work have mostly been conducted on CRAN. Additionally, while there has been some strands of research trying to understand the maintainability of R packages and the issues caused by backward incompatibility, most of them were conducted through surveys and interviews of developers, or through analysis of package metadata. Even though backward incompatibility in R packages remains a point of interest for quite some time, we did not find any existing tools that could statically analyse 2 versions of an R package and identify syntactic breaking changes between them.

3.1 API Evolution and Stability

3.1.1 API Evolution

While API changes have been quite extensively studied in the software engineering literature, most of them were focused on Java and .NET based software ecosystems. Dig and Johnson [22] explored the idea of understanding API changes to provide a foundation for version migration tools that can make it easier for clients to update their APIs. In their study, they

found that 80% of the changes that broke client systems were refactorings. However, their study was focused on 5 large scale Java client projects and their conclusions were based on these case studies. Raemaekers et al. [38] studied 140 clients of the Apache Commons librray to present 4 stability metrics that can have an effect on method changes and removals. Their methodology mostly focused on studying Maven build files and API usage frequency. Jezel et at.[29] studied the binary compatibility between versions of 109 Java programs and 564 releases. The authors went on to conclude that API backward incompatibility is fairly common in version upgrades, however, they don't seem to affect a lot of clients in the context of Java. Thung et al.[14] tried to understand why developers create breaking changes in their API. They interviewed developers from 3 ecosystems, Eclipse, R/CRAN and Node. Their findings revealed that attempts in reducing technical debt, increasing API performance and bug fixes were the primary reasons behind API breaks in the 3 ecosystems. McDonnell et al.[35] look at Andoird APIs and state that they evolve faster than client migration. On a different context, Linares-Vasquez[33] analyze an increase in StackOverflow questions occurs adjacent to major API changes. They also go on to show that developers of android applications have an increase in their activity whenever they encounter changes in API.

3.1.2 API Deprecation

In the area of API deprecation, Robbes et al.[40] study the effect of deprecation in a SmallTalk software system. They conclude that API deprecations can have large impacts on clients and that existing deprecation alert messages generally do not help the situation much. Following up on the same context, Hora et al.[27] studied deprecation messages and found that clients do not generally respond to API changes even though a large amount of them are affected. Raemaekers et al.[39] concluded from their study that developers don't usually follow proper deprecation policies. They found that APIs are commonly removed in

version upgrades without prior deprecation tags, while APIs with deprecation tags stay in the repositories for a long time. Brito et al.[15] try to help this situation by proposing a tool that helps developers with improved deprecation message recommendations.

3.2 The R Ecosystem

German et al.[24] studied the evolution of R ecosystems by looking at the characteristics of core and user-contributed R packages. They studied the growth of packages in terms of size and the community structure around them, to conclude that R ecosystems are rapidly flourishing that requires maintenance to remain healthy. Ooms[37] studied the dependency management issues in CRAN and proposed some future directions through which packaging systems can be modernised to enable dynamic contribution in development without compromising on the reliability of depending on packages. Decan et al. [20] explored the distribution of R packages in the ecosystems like CRAN, Bioconductor and Github and tried to see if there were any meaningful relationships or replication. One of their findings was the Bioconductor and CRAN have a very small number of packages common between them. Then they went on to study the maintainability of CRAN packages and explored topics like the source of errors in CRAN pacakges, as reported by the R CMD Check Tool when a package is being released [17]. These errors were mostly related to the quality of the packages released as per the CRAN standard. Finally, they looked at the inter-repository package dependency issues in R, mostly the relationship between packages released in CRAN and the same development versions of the same packages in Github[21]. They find that development versions in Github could use with more quality check tools such as the R CMD Check to be more reliable in their applications. Finally, Bogart et al. [14] conducted a study on why developers create breaking changes in their package updates. Their methodology involved

interviewing a number of developers, a portion of which were CRAN maintainers. Their findings highlighted that technical debt, efficiency of packages and bug fixes were some of the primary reasons behind creating breaking changes.

The body of research by Decan et al.[20][17][21] was perhaps the closest to our work in the context of checking the health of R ecosystems. However, their work mostly focused on the package release errors and inter-repository dependency issues of CRAN and did not particularly look into the issue of breaking changes and semantic versioning. The R CMD Check tool that they used for their methodology also checks for the quality of a package release based on CRAN rules. It does not necessarily look for semantic versioning issues or documentation of breaking changes in package updates.

3.3 Client Migration Support Tools

Quite a few tools and approaches have been suggested to help clients deal with API breaking changes when migrating between library versions. Kim et al.[31] proposed an approach to identify migration rules from structural changes such as function signature modifications. They also proposed LSDiff[30] two compute differences between two library versions for Java programs. Nguyen et al.[36] presented a graph-based technique to provide API usage suggestions for Java developers. Henkel and Diwan[26], an approach to capture and provide historical refactoring data to developers. Hora et al.[28] presented an approach that involved mining import statement changes to track API evolution. Dagenais and Robillard[19] proposed a recommendation system for API replacements based on changes in the library. On this same context, Schafer et al.[41] proposed a recommendation system that works by mining API usage in client applications. Wu et al.[47] proposed an approach that provides developers with API evolution rules based on text similarity and call dependency graphs.

Our work focuses on detecting breaking changes between two library versions written in R.

3.4 Breaking Change Detection Tools

Most of the breaking change detection tools currently exist online are for strongly typed languages. For .NET code bases, Endjin[6] attempts to detect API changes and correctly determine semantic versioning. LibCheck[7] is another tool for .NET libraries created by Microsoft to detect changes in public interfaces between versions. Other such tools include RevAPI[11] for Java and Swagger[12] for Ruby. For dynamically typed language, availability of such tools seem to be a little more scarce. Cracks [4] focuses on detecting breaking changes in Javascript projects through dynamic analysis by running the existing test suites. Kraaijeveld[32] also talk about a breaking change detection tool for Javascript. For their tool, they use client projects to differentiate between private and public APIs in a library.

Chapter 4

Methodology

In this section, we will first discuss our data acquisition techniques and then move on to talk about the scope, design and functionalities of our tool.

4.1 Data Acquisition

Bioconductor currently consists of 2140 software packages at the time of writing this paper. For the purpose of this study, we investigate the source code of all the release versions of 100 most downloaded packages.

Bioconductor also provides users with access to it's browsable code repository. For any given package, the repository consists of the source code of the current release of the package along with all it's previous releases, represented as git branches. We use node scrapy[34] to crawl through the html pages and get all the release version links.

Once we have all the package names and their available release version numbers, we can download the zip file that consists of that version's source code using the URL pattern :
http://code.bioconductor.org/browse/[package_name]/zipball/[release_version]
where [package_name] and [release_version] are variables that correspond to the name of the package and it's release version respectively.

4.2 A Breaking Change Detection Tool for R libraries

4.2.1 Scope of the Tool

API breaking changes can be broadly defined into two categories : syntactic breaking changes and semantic breaking changes. Syntactic breaking changes mostly include changes in the function signature, that can generally be captured through static analysis. Semantic breaking changes refer to changes in variable types and runtime behaviour. In order to capture semantic breaking changes, dynamic analysis of the code is usually required, which in turn requires extensive test suites for the libraries that cover all the public APIs. We explored the avenue of dynamic code analysis for R projects but found that most packages don't have the necessary test suites to run the APIs in order to extract information from them. Therefore, we decided to move forward with capturing just the syntactic breaking changes with our tool, as the underlying belief was that syntactic changes might still give us a very good picture of the breaking changes that occur in R projects.

In that context, we categorize syntactic breaking changes into three types.

1. **Function Removal :** Where a function is removed in the newer API. In this case, if a function is renamed, our tool captures it as a function removal as well.

2. **Non-default Parameter Modifications :** This refers to adding parameters in the function signature without providing any default values. Such an API change is guaranteed to break client code upon a library update.

3. **Minor Parameter Modifications :** Changes such as parameter renaming, parameter re-ordering or changing the default values of parameters can cause a breaking change depending upon the implementation style of the functions. Since there's a

chance that these types of changes in the function signature might not break the client code in a good number of cases, we categorize them as minor parameter modifications.

Throughout the rest of this paper, we use the terms API and functions interchangeably.

4.2.2 Tool Overview

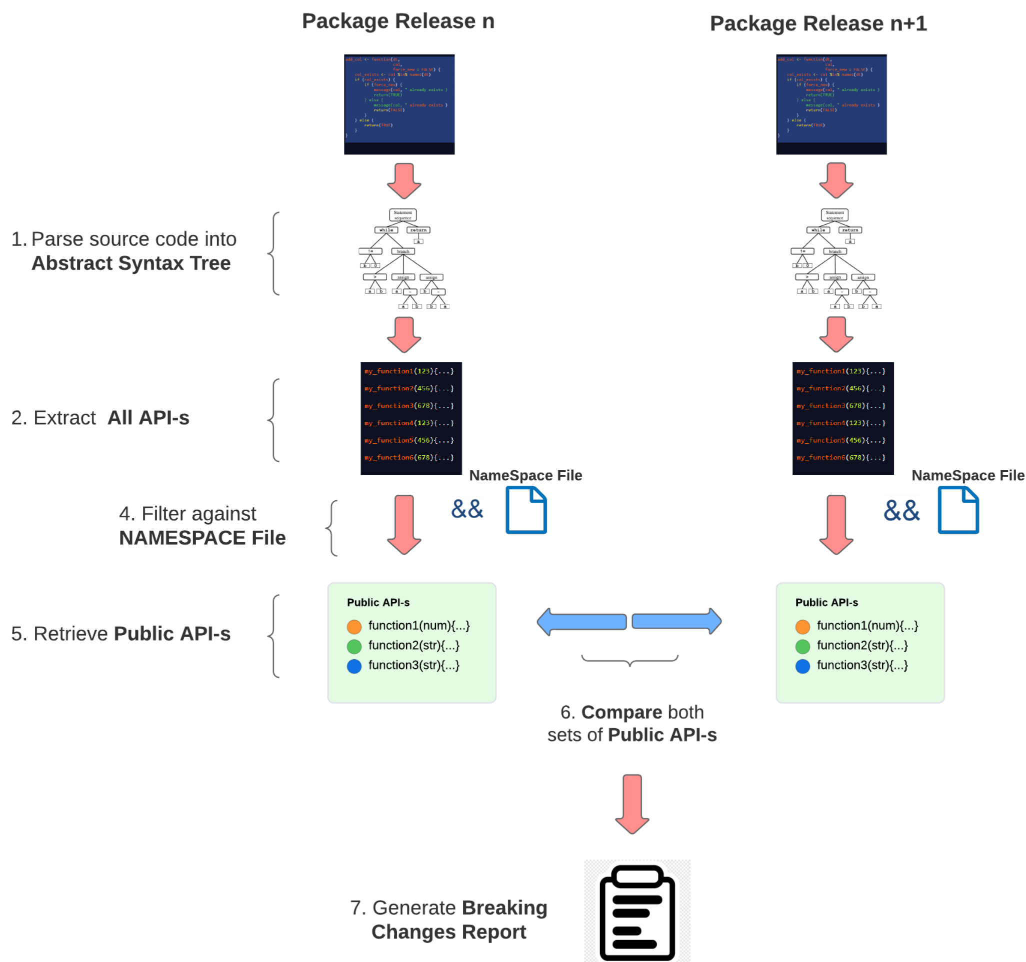

Figure 4.1: Overview of the API breaking change tool

In order to conduct our study of breaking changes in Bioconductor packages, we create a tool that uses static analysis of the source code to detect breaking changes between two package versions written in R. Figure 4.1 provides an overview over our tool's mechanisms. The utility of this tool can be extended to analyse R packages in most modern R code repositories such as CRAN or standard Git sources.

As for the approach behind our tool's mechanism, given two release versions of a package, V_n and V_{n-1}, where n represents the release version V of a package, for every release we first parse all the R code from the release source files and extract all the function signatures. We then filter out all non-public functions and only retain the public function signatures that the package developers created for client use. This leaves us with 2 sets of public APIs, one for each package release. We represent each api as a model that consists of the function signature and the class it may belong to. We then compare the two sets of APIs to look for any syntactic breaking changes that may exist between the APIs of the two releases.

4.2.3 Parsing the R source code

In order to perform static analysis on the source code of the packages, we wanted to parse the R code in a manner that would make it easy for us to navigate through the code and identify the right patterns. We use a tool called xmlparsedata [18] to generate XML tree representations of the R source code we find in the packages. Such an XML tree representation provides us with an organized view of the structure of R expressions and functions. For instance, the XML tree representation of the code in figure 4.2 would be the XML document in figure 4.3.

We then go on to convert the XML tree into an abstract syntax tree (AST) which is represented as a JavaScript object using xml-tools/ast [13] where the nested XML elements

```r
sampleFunction <- function(a,b){
a+b
}
```

Figure 4.2: Sample Function Declaration in R

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<exprlist>
  <expr>
    <expr>
      <SYMBOL>sampleFunction</SYMBOL>
    </expr>
    <LEFT_ASSIGN>&lt;-</LEFT_ASSIGN>
    <expr>
      <FUNCTION>function</FUNCTION>
      <OP-LEFT-PAREN>(</OP-LEFT-PAREN>
      <SYMBOL_FORMALS>a</SYMBOL_FORMALS>
      <OP-COMMA>,</OP-COMMA>
      <SYMBOL_FORMALS>b</SYMBOL_FORMALS>
      <OP-RIGHT-PAREN>)</OP-RIGHT-PAREN>
      <expr>
        <OP-LEFT-BRACE>{</OP-LEFT-BRACE>
        <expr>
          <expr>
            <SYMBOL>a</SYMBOL>
          </expr>
          <OP-PLUS>+</OP-PLUS>
          <expr>
            <SYMBOL>b</SYMBOL>
          </expr>
        </expr>
        <OP-RIGHT-BRACE>}</OP-RIGHT-BRACE>
      </expr>
    </expr>
  </expr>
</exprlist>
```

Figure 4.3: XML tree representation of a function

are represented as nodes with parent-child relationships that can be easily traversed. The reasons behind the choice of using JavaScript as the programming language and NodeJS as the framework for the tool's development include the fact that it can be easily extended to a web application that can be used by both R library developers and clients, and our team's experience with JavaScript development.

4.2.4 Function Extraction

As previously mentioned, R functions can be declared in quite a few ways. These include the traditional S3 mode of function declaration, as demonstrated in sampleFunction1, and the S4 modes of function declaration, as showin in sampleFunction2 and sampleFunction3 in figure 4.4. For S4 function declarations, it is also possible to use named parameters, as shown in the setReplaceMethod call for sampleFunction3, where the signature is provided with a named parameter. We wrote pattern matching algorithms where we traverse the AST and extract all such possible patterns of function declarations. For every function extracted, we create a function model and represent it as a digital object, or more specifically, a JavaScript object that consists of the function name, parameters, the function body, the function signature or the class of object that function expects as it's parameter, the parent class the function may belong to and finally how the function is represented in the NAMESPACE file of the package. An example of a function model created from sampleFunction3 in figure 4.4 is shown in figure4.5.

Additionally, another practice that developers follow is declaring an R function first in the S3 mode and then creating an S4 declaration of the same function, either with the same name or a different name, as shown in the example in figure 4.6. In such cases, the sampleFunction5 doesn't have a function signature or body in it's declaration since it is using the properties and

functionality of sampleFunction4. To handle these cases, we first extract all the functions, identify all such S4 declarations, find the S3 declarations they are referring to and fill out the properties in our function models. This is also the reason we don't perform any prior filtering in our initial extraction of all functions present in the source code of the packages. In summary, once our function extraction stage is complete, for releases V_n and V_{n-1} of a package, we get two sets of function models $ALL_FUNCTIONS_MODEL_SET_n$ and $ALL_FUNCTIONS_MODEL_SET_{n-1}$.

```
sampleFunction1 <- function(a,b){
a+b
}

setMethod("sampleFunction2", "testClass", function(a){
a})

setReplaceMethod("sampleFunction", signature= "testClass", function(x){
return x})
```

Figure 4.4: S3 and S4 function dispatch methods

```
{
"name" : "sampleFunction3",
"parameters" : ["x"],
"body" : "return x",
"signature" : "testClass"
"parentClass" : null,
"representation" : "sampleFunction3<-"
}
```

Figure 4.5: S3 and S4 function dispatch methods

```
sampleFunction4 <- function(str){
toupper(str)
}

setMethod("sampleFunction5", "character", sampleFunction4)
```

Figure 4.6: S4 function declared using an S3 function

4.2.5 Public API Filtration

API breaking changes refer to the differences in the client facing API-s, which means we have to filter out all private functions from our sets of extracted function models for both packages being compared. In general, for a given R package, the developers can specify what functions they want to make available to the public in the packages NAMESPACE file. Developers can utilize commands such as export(), S3method() and exportMethods() in quite a few ways to make their functions available to the public. In addition, modern software such as roxygen[46] can automatically craete the NAMESPACE file for an R package as long as the developers annotate the functions they want to export as public APIs in their source code. Some example NAMESPACE file commands are shown in figure 4.7 where the functions exported as public APIs are window.Vector(), aggregate.Vector(), aggregate.Rle(), aggregate.List(), nrow(), ncol() and dim().

Since there are textual variations in how the commands are declared in the NAMESPACE file of an R package, we create our own NAMESPACE file parser that uses regular expressions to read all the exported function names. Once we extract all the function names from the NAMESPACE file, we filter out out the private functions from our

$ALL_FUNCTIONS_MODEL_SET_n$, leaving us with a function model set of only the

public APIs, $PUBLIC_API_SET_n$ for a release version of a package V_n.

```
S3method(window, Vector)
export(
    aggregate.Vector,
    aggregate.Rle)
export(aggregate.List)
exportMethods(nrow, ncol, dim)
```

Figure 4.7: Public function declarations in the NAMESPACE file

4.2.6 Comparing Public API Models And Reporting Breaking Changes

Once we have both sets of public API models for two release versions of a package, we compare them to identify the syntactic breaking changes. As previously mentioned, our tool categorizes syntactic breaking changes into 3 different types. For any API available in $PUBLIC_API_SET_{n-1}$ but not available under the same name and class or at all in $PUBLIC_API_SET_n$, we categorize it as a function removal. We ignore cases where a function was deprecated or defunct for at least 1 version before it's removal as it abides by the guidelines of Bioconductor and also generally makes the client aware that the API is going to be removed with a prior notice. If an API exists in both release versions but a parameter without a default value is added or removed from the function signature of the new release, we count the API as a major parameter change. If the number of non-default parameters remain the same between an API in the two versions but parameters are renamed, reordered, default parameters are added in the middle of the sequence or removed, we count the API as an implementation sensitive parameter change.

Chapter 5

Results

5.1 RQ1 : How frequently are package releases backward incompatible?

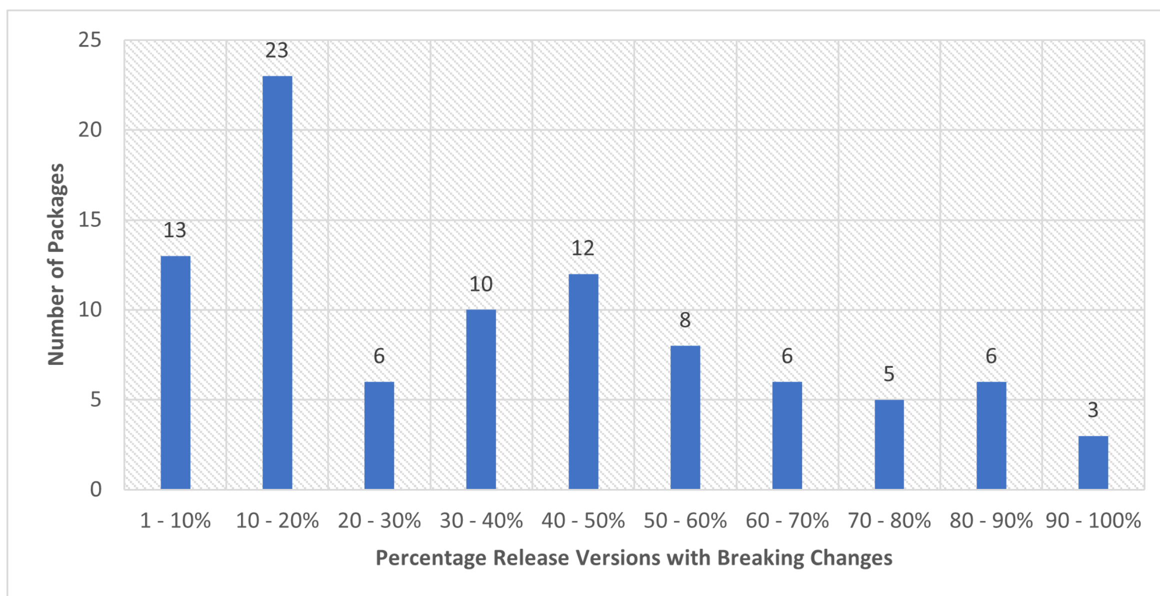

Figure 5.1: Number of packages against percentage release versions that consist of a breaking change

The goal of this research question was to understand how widespread the issue of breaking changes are in the Bioconductor ecosystem. To answer this, we compare every two consecutive release versions of a package for breaking changes using our tool. We consider

	Total	Contains B.C	Contains S.V.I
Releases	1680	471	377 (80%)

Table 5.1: Releases with Semantic Versioning Issues

Package Name	No. of Releases
BiocIO	2
MatrixGenerics	3
rhdf5filters	3
rhdf5lib	7
ScaledMatrix	1
biomformat	10
Rhtslib	12

Table 5.2: Package with no breaking changes and their version numbers

releases that consist of 1 or more breaking changes we previously defined to be backward incompatible.

In the 100 packages we study, we find a total of 1680 releases with breaking changes occurring in 471 of them indicating that on average, breaking changes occur in around every 3.5 releases. Figure 5.1 gives us a deeper look at the distribution of releases being backward incompatible. Here we see that apart from 8 packages, all the others consist of breaking changes in their releases to some extent or the other. Additionally, we see that 28 packages consisted of breaking changes in over 50% of their releases. This indicates that a client is likely to see a breaking change in every other release of these 28 packages or sooner.

The 8 packages with no backward incompatible releases are listed in table 5.2. One insight that stands out in that context is their number of releases these packages have. While the top 100 packages seem to have over 14 releases on average, these 8 packages have an average of 4.75 release versions.

5.2 RQ2 : What is the distribution of the different types of breaking changes across the packages?

Breaking change type	Count
Removed Functions	2056
Non-default parameter modification	572
Minor parameter modification	801
total	3429

Table 5.3: Types of breaking changes and their count across all packages

The goal of this research question was to understand what types of breaking changes developers introduce the most to raise awareness amongst the Bioconductor community. During the experiment for RQ1, we also record the three different types of breaking changes that occur in the releases. As shown in table 5.3, we find 3429 breaking changes across the 100 packages in total, of which 2056 were function removals, 572 were non-default parameter modifications and 801 minor parameter modifications. In figure 5.2 we represent the different types of breaking changes that occur in total numbers in the entire life span of a package for all the 100 packages as box plots. We see that function removals is the most common type of breaking change that developers make pretty consistently across all packages, with a median of 6.5 and a mean of 20.9 for a package's life span. Minor parameter modifications, with a mean of 8.1 and a median of 2, and non-default parameter modifications, with a mean of 5.8 and a median of 1, follow as the second and third types respectively.

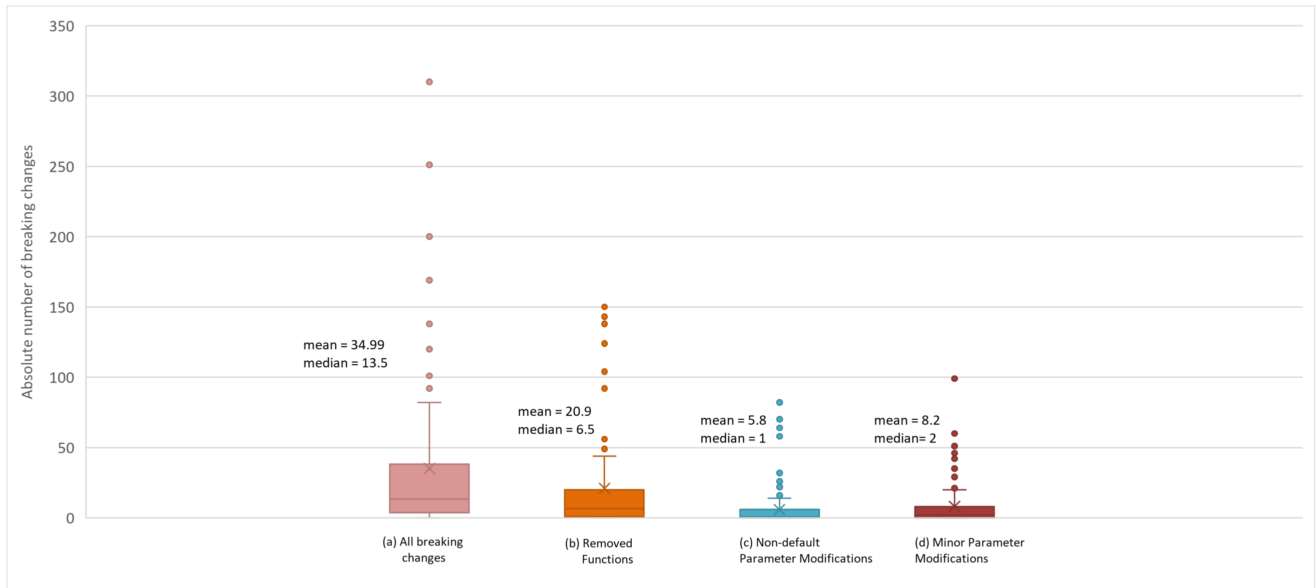

Figure 5.2: Number of (a) all breaking changes (b) function removals (c) non-default parameter modifications (d) minor parameter modifications that occur across all releases of a package

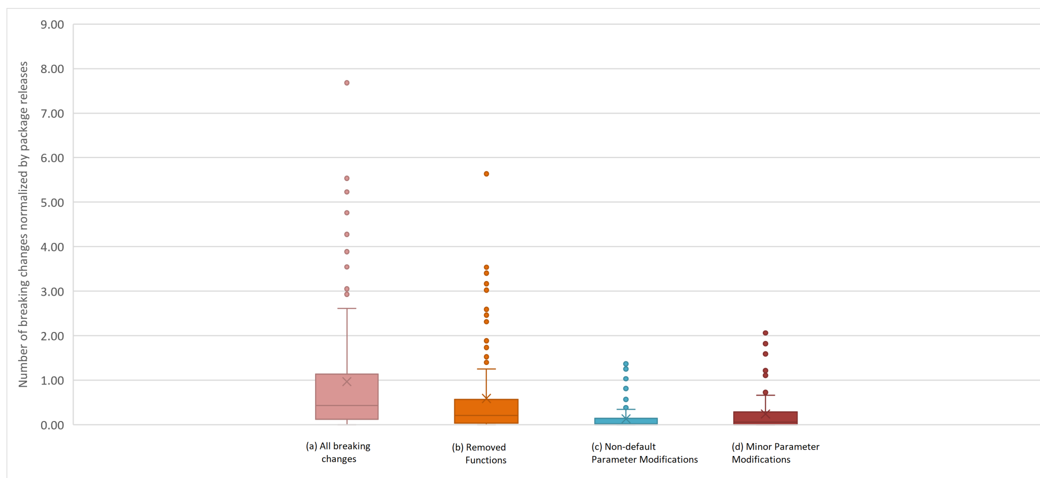

Figure 5.3: Number of (a) all breaking changes (b) function removals (c) non-default parameter modifications (d) minor parameter modifications for every package normalised by the number of releases

Since packages can have varying number of releases and a package with a large number of releases may have a large number of breaking change of a certain type, we also present figure

5.3, where we normalized the breaking change numbers by the number of releases for every package. The normalized figures also confirm our finding that function removals are the most common type of breaking change.

5.3 RQ3 : How often do developers have semantic versioning issues when releasing a new version?

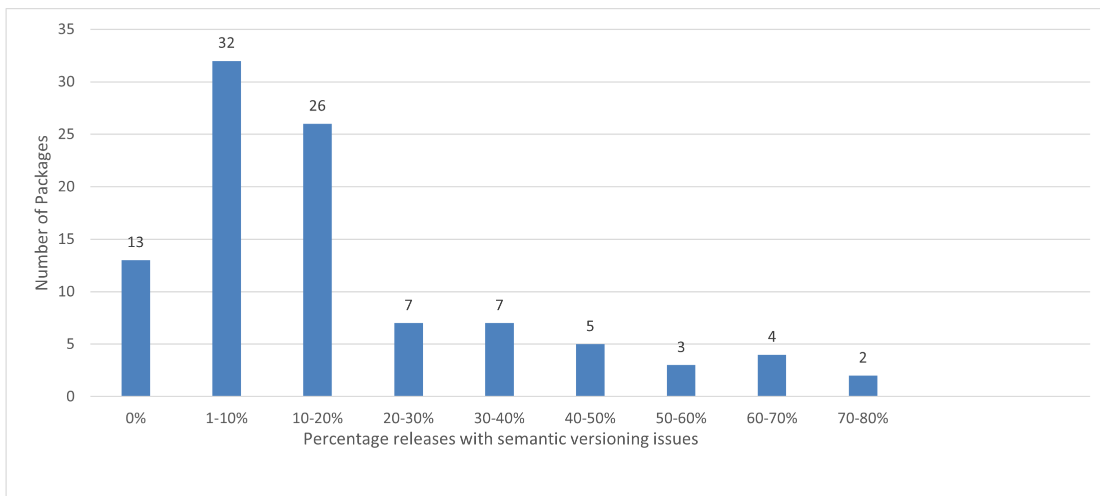

Figure 5.4: Distribution of packages with semantic versioning issues in their percentage releases

Maintaining semantic versioning in package releases allows clients to know what packages may have major breaking changes in them. This is generally done by updating the major version of a package if it contains a change that can break existing client code. For this research question, along with the differences in the API signatures between two consecutive releases, we also extract the release version of the package by parsing the DESCRIPTION file. We look at the 1680 releases and find 377 semantic versioning issues, which is, a major breaking change was in the release but the major version of the release was not updated.

Figure 5.4 shows the distribution of semantic versioning issues in the percentage releases across the packages. We see that 87 of the packages of semantic versioning issues in all their releases to some extent or the other. 21 packages have semantic versioning issues in more than 30% of their releases.

However, figure 5.4 shows us the issues for all releases, both that consist of breaking changes and those that do not. A closer look at the issue tells us a slightly more concerning story about developer awareness when maintaining semantic versioning. Figure 5.5 shows the distribution of semantic versioning issues for only releases that had a major breaking change. In this diagram, we see the distribution shift towards higher numbers in most percentage clusters, with 8 packages not updating their semantic versioning for 100% of the releases that had a major breaking change.

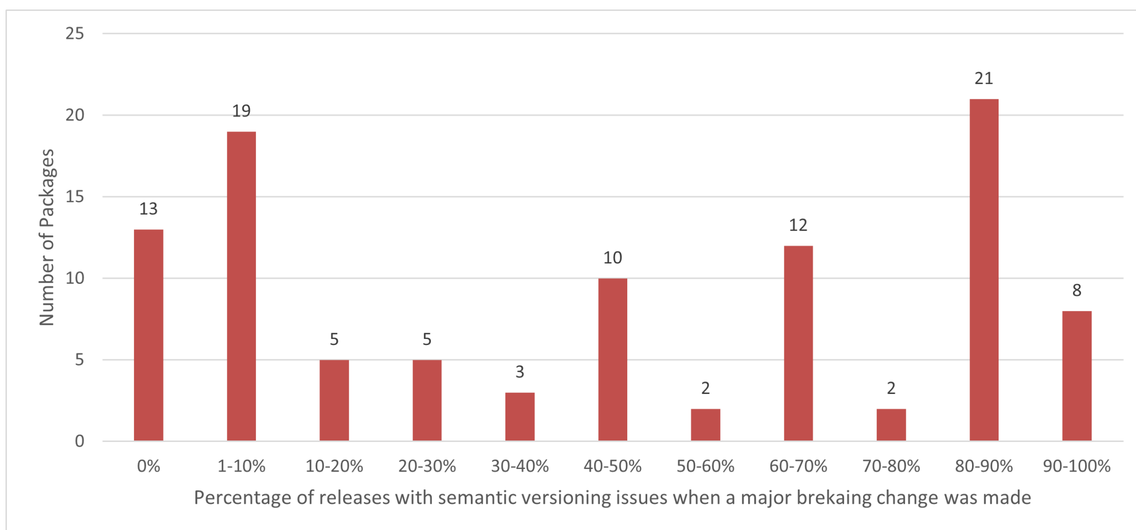

Figure 5.5: Distribution of packages with semantic versioning issues in their percentage releases that consisted of a breaking change

5.4 RQ4 : To what extent do developers document breaking changes?

Breaking change type	Detected	Undocumented
Removed Functions	1406	748(53.2%)
Non-default parameter modification	369	205 (55.6%)
Minor parameter modification	474	298 (62.9%)
Total	2249	1251(55.6%)

Table 5.4: Types of undocumented breaking changes

Documenting breaking changes properly allows clients to get a prior understanding of what they may need to adjust in their code. The goal of this research question was to see how often API breaking changes go undocumented. Here, we manually inspect all the breaking changes documented in a package's NEWS.md file against all the syntactic breaking changes our tool detected. We investigated the top 50 most downloaded packages, since comparing the log files generated by our tool against the NEWS files of every package was fairly time consuming and looking at all 100 packages was not practical due to time constraints.

In the 50 packages we observed, our tool detected 2249 breaking changes in total, of which only 998 were documented in the NEWS.md file, meaning over 55% of the breaking changes went undocumented on average. A closer look at the situation in figure 5.6 shows function removals go as the most undocumented type of breaking change, with packages having a mean of 51.10% and a median of 56.07% undocumented function removals. Minor parameter modifications follow with a mean of 36.35 and a median of 45.00%. Finally, non-default parameter modifications have a mean of 25.62% and a median of 0. This is perhaps due

to the fact that a lot of the packages don't have non-default parameter modifications at all, and for those that do, developers seem to be very aware of this type when writing the documentation.

While it's understandable that developers don't always document minor parameter modifications, the case of function removals may be explained in the sense that developers remove a lot of APIs that they deem to be redundant or not used by clients much. In that regard, perhaps they don't feel like documenting their removals either.

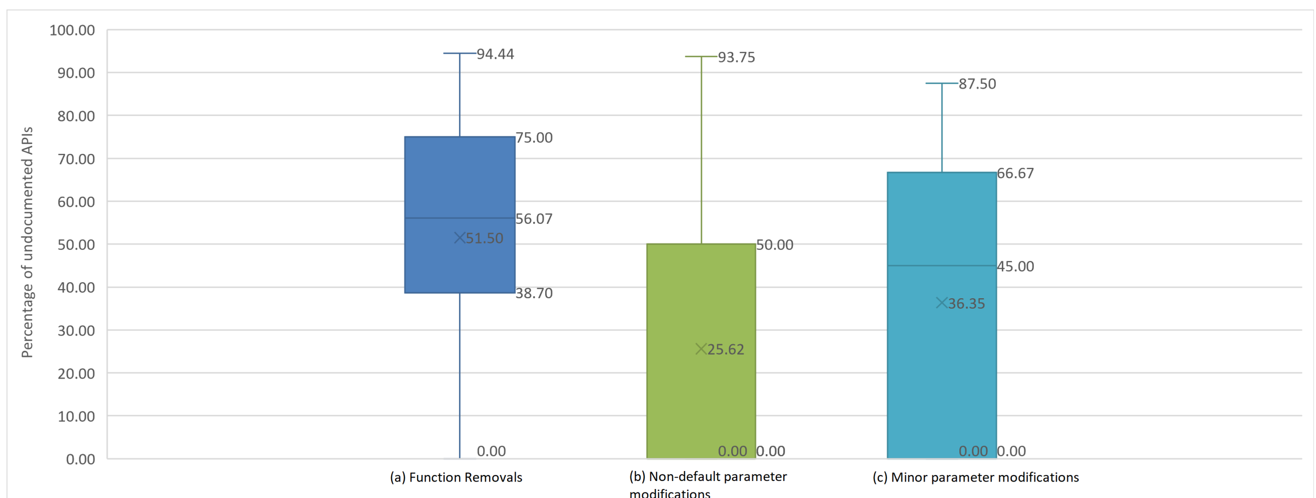

Figure 5.6: Percentage of undocumented breaking changes of different types across the 91 packages

5.5 RQ5 : How frequently do broken APIs occur in client projects?

Library	Bioc Release Jump	Total F.R	F.R. Found in Projects	Total Dependent Projects
IRanges	2_14 to 3_1	112	0	0
Scater	3_5 to 3_6	79	5	21
AnnotationDbi	2_4 to 2_5	78	0	0
GenomicRanges	2_13 to 2_14	60	0	0
HDF5Array	3_4 to 3_5	52	0	0
Biostrings	2_2 to 2_3	40	0	0
ggtree	3_4 to 3_5	34	1	3
Dose	3_2 to 3_3	28	2	4
clusterProfiler	3_2 to 3_3	25	2	2
complexHeatmap	3_8 to 3_9	23	2	1

Table 5.5: Breaking changes and their impact on client projects

For this research question, we study 10 releases that had the most function removals and try to understand how frequently API breaking changes actually affect client projects. Since no large scale dataset of R projects were readily available, we manually inspected the present of each removed function in the R projects hosted on Github, using Github's code search mechanism. This is also the reason why we excluded parameter modifications from our study, since it is very difficult and time consuming to check for presence of parameters in the functions called by clients just by reading through code, especially since a large majority of function calls allow variable numbers of parameter implementation. Table 5.5 shows the 10 packages, the release upgrades we studied, the total number of function removals in those releases, the number of those removed functions we found in client projects and finally the number of client projects we found relying on those API-s.

In total, we see 512 functions removed in the 10 releases and we find 12 of these API-s used by 1 or more projects. This is about 2% on average. However, we also find 31 projects affected by these changes from just 10 version upgrades, which is a considerable number given that we found 1680 releases in the top 100 most downloaded packages. It is also clear that some API breaks affect more projects than others given their popularity with Scater's release being a good example where 5 breaking changes affected 21 projects. Finally, it should be noted that these are some very conservative numbers, given that a large number of projects perhaps don't get uploaded to github at all, and thereby, were not a part of our study.

For the 31 projects affected, we also look at how the API usage could be fixed. For 24 of the projects, we found that a single line fix could be made in the code where the API could be directly replaced by either another API in the same library, because the original API was renamed into the new one, or by another API from a different library, because the developers removed the API from their package since it was available elsewhere and they wanted to avoid code duplication. For example, we found a project using the function enrichMap() from the package clusterProfiler version 3.2. However, from the next release of clusterProfiler, the function was removed from the library and was included in another library called enrichPlot. In this case, the authors would just update their code to the right library name. For the other 7 projects, we found that the functions were refactored to be an attribute of a parent class and an object would have to be declared for the functions to work.

Chapter 6

Discussion

From RQ1, we see that backward incompatible releases in packages are quite popular in the 100 Bioconductor libraries we studied. In fact, apart from just 8 packages, which have comparatively fewer releases in the first place, all other 92 packages have backward incompatibility in their releases to some extent or the other. The notable insight was perhaps that we found 40 packages that had breaking changes in over 40% of their releases. This can be a cause of concern given that clients are likely to use multiple packages in their code and trying to upgrade all their packages at the same time increases the likelihood for having to deal with breaking changes even more. What does not help the scenario is Bioconductor's package release policy. Generally, a new version of Bioconductor comes out twice annually with all it's packages having a release for that version of Bioconductor. Given the default behaviour of Bioconductor's package installation tool, clients are likely to run a command that updates all packages used in their code to the latest version of Bioconductor. While this issue can technically be overridden by manually choosing package versions to be installed instead of upgrading all packages, most generic users will find the choice between having to deal with breaking changes or putting in extra effort on specifying release versions of the packages they use time consuming and frustrating.

RQ2 lends us insight on the types of breaking changes that occur the most in the ecosystem. Here we see that function removals are the most dominant type of breaking change in terms of frequency, with minor parameter modifications following up as a close second and non-

default parameter modifications afterwards. In that context, parameter modifications are quite understandable and perhaps even unavoidable in certain circumstances. In fact, Bogart et al. [14] mention that optimization and better performance are some of the primary reasons behind API refactoring and it's quite intuitive how adding parameters or making changes in their default values may be a part of that process. In case of function removals however, some of the potential reasons we have noticed through our preliminary manual inspection include attempts to reduce code duplication in the overall repository, since other libraries may be providing the same functions, changing the name of the function to adhere to certain naming styles or just removing the function as a public API because the developers deem that clients do not use these functions frequently. If developers plan out the early phases of their releases and the structure of their APIs better, perhaps this whole sale approach of removing numerous functions at a later point can be avoided.

To help clients be aware of breaking changes in their library versions, developers are supposed to maintain semantic versioning. Looking at RQ3, we see that of the 471 releases with backward incompatibility, 377 of them have semantic versioning issues, which is roughly around 80% of the releases. Additionally, results of RQ4 tell us that developers do not document breaking changes around 54% of the times. Whether this happens due to negligence or lack of awareness of breaking changes created in the releases is unclear but these are definitely issues that should be looked upon both by library developers and by the software repository maintenance committee. Developers can perhaps use API analysis tools like ours to get a comprehensive view of any breaking change they may have missed out on while documenting. On the other hand, the Bioconductor maintainers can enforce stricter rules over documentation and proper versioning before every release cycles.

In RQ5, we manually inspect 10 package releases and the effect of the removed functions on client projects. In that aspect, we find 12 of the 512 removed functions implemented in

actual projects. While this constitutes of only 0.02% of the removed APIs, we see that the number of projects affected by these changes are 30. This is quite a large number considering that there are 1680 releases in our data set. Given that our methodology involved looking at projects hosted on Github, there may be many other client projects that are impacted by these breaking changes that were not a part of our inspection. Since breaking changes do seem to occur in projects, users are recommended to upload their projects along with the RENV lock file. This keeps track of all the package versions used in an R project. This will make it easier for the R research community in general to reproduce the results of older code without having to upgrade all package versions involved in a project.

Chapter 7

Threats To Validity

The validity of our research is prone to quite a few threats. We start off with internal validity. Our tool is designed to only capture syntactic breaking changes in APIs by analysing only the function signature. Since we ignore semantic breaking changes such as changes in types of entities such as the returned variables or parameters, the actual number of breaking changes in the packages might be higher than the numbers reported in RQ1 and RQ2. Secondly, for RQ4, our results are generated through manual inspection of the package documentation compared with the list of breaking changes created by our tool. This approach is prone to human error since we might have miscounted the number of breaking changes on either sides, especially in cases where the count of breaking changes exceeded hundreds. Thirdly, for RQ5 we manually inspect the present of removed functions in github projects. Since there's a good chance that a large number of projects were not hosted on github but were in fact affected by the breaking changes, we might have underrepresented that number.

External validity refers to the generalizability factor of our findings. Our study was conducted on only the top 100 packages in Bioconductor, whereas the software repository consists of more than 2000 packages at the time of writing this paper. The number and nature of breaking changes may vary significantly in the packages we did not analyse. Additionally, our findings are limited to the Bioconductor repository and only the R language itself. The nature of other software repositories vary in developer best practices, release mechanisms and package structures.

Chapter 8

Future Work

During the preliminary investigations before the development of our tool, we noticed quite a few API diff tools exist for static languages like Java and C++. These tools cover a wide variety of breaking changes including changes in variable types since the languages inherently provide more syntax information through annotations and being strongly typed. This is not true in case of dynamic languages, where type information and public/private function annotations are not available, limiting the range of breaking changes that can be captured through static analysis. With R being a dynamically typed language, during the development of our tool, we also explored dynamic analysis to capture the semantic breaking changes along with the syntactic ones that occur between library releases. However, dynamic analysis approaches require the code to run and this means comprehensive test cases of all public APIs were required to progress in that direction. Unfortunately, we saw that R packages often do not have any test cases at all and in instances that they do, the tests only cover a small fraction of the public APIs. If developers start writing more test suites for their packages, perhaps the avenue of dynamic analysis can be explored better.

The R software repositories only seem to be growing with time as more and more packages are added. Since API breaking changes will inevitably exist in these packages, research and development in not only breaking change detection but also on API migration support tools should be explored for R. A general approach for these migration recommendation tools may involve parsing the client code to identify APIs and then analysing the library source code

to check for breaking changes. Since multiple libraries can have APIs of the same names, identification of the APIs in the client code can be semi automatic, where the client will point out which APIs belong to which libraries.

Chapter 9

Conclusions

We first present a tool that can detect syntactic breaking changes in R packages. We then proceed to conduct a study of backward compatability and breaking changes in the 100 most downloaded packages in Bioconductor. We find that breaking changes exist in 28% of all package releases, with function removals being the most popular type of breaking change. We also find that developers don't maintain proper semantic versioning in 22% of these releases. Additionally, we also manually inspect the documentation of the 50 packages and find that 55% of the with breaking changes in APIs reported by our tool go undocumented. To study the impact of these breaking changes in client projects, we inspect 10 releases with the most function removals in them and find 0.02% of the APIs in 31 client projects. Finally, we go on to discuss the implications of these findings for clients, developers and researchers. We also explore future work for more improving API change detection tools to capture a wider set of breaking changes through dynamic analysis of code.

Bibliography

[1] Bioconductor. https://www.bioconductor.org/.

[2] Bioconductor package support blog post. https://support.bioconductor.org/p/110153/.

[3] Bioconductor package support blog post 2. https://support.bioconductor.org/p/120724/.

[4] Cracks. https://github.com/semantic-release/cracks.

[5] Cran. https://cran.r-project.org/mirrors.html.

[6] An experiment to automatically detect api breaking changes in .net assemblies and suggest a semantic version number. https://endjin.com/blog/2016/02/an-experiment-to-automatically-detect-api-breaking-changes-in-dot-net-assemblies-an

[7] Finding changes between assembly versions with libcheck. https://www.oreilly.com/library/view/windows-developer-power/0596527543/ch04s05.html.

[8] Install older version of a package. https://stackoverflow.com/questions/71247393/problems-installing-older-version-of-bioconductors-mixomics-packages-in-r.

[9] Install specific version of a package. https://stackoverflow.com/questions/49487171/r-how-to-install-a-specified-version-of-a-bioconductor-package.

[10] Install specific version of a package. https://stackoverflow.com/questions/37503884/install-a-specific-release-of-bioconductor-package.

[11] revapi. https://revapi.org/revapi-site/main/index.html.

[12] Using swagger to detect breaking api changes. https://swagger.io/blog/api-development/using-swagger-to-detect-breaking-api-changes/.

[13] xmltools/ast (5.0.5). https://www.npmjs.com/package/@xml-tools/ast, 2021.

[14] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. How to break an api: Cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 109–120, New York, NY, USA, 2016. Association for Computing Machinery.

[15] Gleison Brito, Andre Hora, Marco Tulio Valente, and Romain Robbes. Do developers deprecate apis with replacement messages? a large-scale analysis on java systems. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 360–369, 2016.

[16] John M. Chambers. S, R, and Data Science. *The R Journal*, 12(1):462–476, 2020.

[17] Maëlick Claes, Tom Mens, and Philippe Grosjean. On the maintainability of cran packages. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 308–312, 2014.

[18] Gábor Csárdi. xmlparsedata (1.0.5). https://github.com/r-lib/xmlparsedata, 2022.

[19] Barthélémy Dagenais and Martin P. Robillard. Recommending adaptive changes for framework evolution. *ACM Trans. Softw. Eng. Methodol.*, 20(4), sep 2011.

[20] Alexandre Decan, Tom Mens, Maelick Claes, and Philippe Grosjean. On the development and distribution of r packages: An empirical analysis of the r ecosystem. ECSAW '15, New York, NY, USA, 2015. Association for Computing Machinery.

[21] Alexandre Decan, Tom Mens, Maëlick Claes, and Philippe Grosjean. When github meets cran: An analysis of inter-repository package dependency problems. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 493–504, 2016.

[22] Danny Dig and Ralph Johnson. How do apis evolve? a story of refactoring: Research articles. 18(2):83–107, mar 2006.

[23] Carey V.J. Bates D.M. Gentleman, R.C. et al. Bioconductor: open software development for computational biology and bioinformatics. *Genome Biol 5, R80*, 2004.

[24] Daniel M. German, Bram Adams, and Ahmed E. Hassan. The evolution of the r software ecosystem. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 243–252, 2013.

[25] Mercatelli D. Giorgi FM, Ceraolo C. The r language: An engine for bioinformatics and data science. *Life (Basel)*, 2022.

[26] J. Henkel and A. Diwan. Catchup! capturing and replaying refactorings to support api evolution. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 274–283, 2005.

[27] André Hora, Romain Robbes, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, and Marco Tulio Valente. How do developers react to api evolution? the pharo ecosystem case. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 251–260, 2015.

[28] André Hora and Marco Tulio Valente. Apiwave: Keeping track of api popularity and migration. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 321–323, 2015.

[29] Kamil Jezek, Jens Dietrich, and Premek Brada. How java apis break – an empirical study. *Information and Software Technology*, 65:129–146, 2015.

[30] Miryung Kim and David Notkin. Discovering and representing systematic code changes. In *2009 IEEE 31st International Conference on Software Engineering*, pages 309–319, 2009.

[31] Miryung Kim, David Notkin, and Dan Grossman. Automatic inference of structural changes for matching across program versions. In *29th International Conference on Software Engineering (ICSE'07)*, pages 333–343, 2007.

[32] Michel Kraaijeveld. Detecting breaking changes in javascript apis. 2017.

[33] Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. How do api changes trigger stack overflow discussions? a study on the android sdk. ICPC 2014, page 83–94, New York, NY, USA, 2014. Association for Computing Machinery.

[34] Stefan Maric. node-scrapy (0.5.0). https://www.npmjs.com/package/node-scrapy, 2020.

[35] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of api stability and adoption in the android ecosystem. In *2013 IEEE International Conference on Software Maintenance*, pages 70–79, 2013.

[36] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. A graph-based approach to api usage adaptation. *SIGPLAN Not.*, 45(10):302–321, oct 2010.

[37] Jeroen Ooms. Possible directions for improving dependency versioning in r. *R J.*, 5:197, 2013.

[38] Steven Raemaekers, Arie van Deursen, and Joost Visser. Measuring software library stability through historical version analysis. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 378–387, 2012.

[39] Steven Raemaekers, Arie van Deursen, and Joost Visser. Semantic versioning versus breaking changes: A study of the maven repository. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 215–224, 2014.

[40] Romain Robbes, Mircea Lungu, and David Röthlisberger. How do developers react to api deprecation? the case of a smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, New York, NY, USA, 2012. Association for Computing Machinery.

[41] Thorsten Schäfer, Jan Jonas, and Mira Mezini. Mining framework usage changes from instantiation code. ICSE '08, page 471–480, New York, NY, USA, 2008. Association for Computing Machinery.

[42] R Core Team. R-manuals. https://cran.r-project.org/manuals.html.

[43] R Core Team. Writing r extension. https://cran.r-project.org/doc/manuals/r-release/R-exts.html.

[44] S Tippmann. Programming tools: Adventures with R. *Nature*, 517:109–110, 2015.

[45] Homer White. 14.1 programming paradigms | beginning computer science with r. https://homerhanumat.github.io/r-notes/programming-paradigms.html.

[46] Hadley Wickham, Peter Danenberg, Gábor Csárdi, and Manuel Eugster. roxygen2: In-line documentation for r. https://roxygen2.r-lib.org/, 2022.

[47] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. Aura: a hybrid approach to identify framework evolution. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 325–334, 2010.