University of California

Los Angeles

Challenges of Scaling Embedded Scientific Computing

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Statistics

by

Jeroen C.L. Ooms

2014

Abstract of the Dissertation

Challenges of Scaling Embedded Scientific Computing

by

Jeroen C.L. Ooms

Doctor of Philosophy in Statistics

University of California, Los Angeles, 2014

Professor Mark Hansen, Chair

foo bar

The dissertation of Jeroen C.L. Ooms is approved.

Mark Handcock

Deborah Estrin

Jan de Leeuw

Mark Hansen, Committee Chair

University of California, Los Angeles

2014

To Elizabeth . . .

who never gave up trying to understand what it is that I do

Table of Contents

List of Figures

List of Tables

Acknowledgments

This research would not have been possible without the help of many people. I would like to thank my adviser Mark Hansen and other committee members Jan de Leeuw, Deborah Estrin and Mark Handcock for giving me the confidence, support and liberty to pursue this somewhat unconventional interdisciplinary research. Their blessing and advice provided the perfect balance between guidance and creative freedom that allowed me to get the most out ouf my doctoral research. I thank Jan specifically for welcoming me at UCLA and personally making my California adventure possible. I am also grateful to my colleagues at Mobilize and OpenMHealth. Collaborating with these professional software developers was a big inspiration to my research and I thoroughly enjoy working with each of the individual team members. Finally a warm hug goes to our student affairs officer Glenda Jones for going above and beyond her duties to take care of the statistics students at UCLA.

Then there are the countless members of the `R` community that generously provided valuable support, advice, feedback and criticism. In particular, much of the initial research builds on pioneering work and advice from Jeffrey Horner. Expertise from Dirk Eddelbuettel and Michael Rutter provided important foundations for developing software systems with `R` on Debian and Ubuntu systems. The RStudio folks, including Hadley Wickham, JJ Allaire, Joe Cheng and Yihui Xie have given lots of helpful technical support. Finally I would like to thank the many individuals and organizations that adopted the `OpenCPU` software, especially in the early stages. By investing their time and energy in my work they were an important source of feedback and motivation to keep improving the software.

Vita

1974–1975	Campus computer center "User Services" programmer and consultant, Stanford Center for Information Processing, Stanford University, Stanford, California.

Publications

MADHOUS Reference Manual. Stanford University, Dean of Student Affairs (Residential Education Division), 1978. Technical documentation for the MADHOUS software system used to assign students to on-campus housing.

CHAPTER 1

The Changing Role of Statisticians and their Software

1.1 The rise of data

The field of computational statistics finds itself in a transition that is bringing tremendous opportunities but also challenging some of its foundations. Historically, statistics has been an important early application of computer science, with numerical methods for analyzing data dating back to the 1960s. The discipline has evolved largely on its own terms and has a proud history with pioneers such as John Tukey using some of the first programmable machines at Bell Labs for exploratory data analysis (Brillinger, 2002). Well known techniques such as the *Cooley-Tukey FFT algorithm* (Cooley and Tukey, 1965) and the *box-plot* (Tukey, 1977) as well as computer terminology such as *bit* and *software* were conceived at this time. These visionary contributions to the process of computing on data laid the foundations for current practices in data analysis. Computer programs developed at Bell Labs in the 1970s and 1980s, such as `UNIX` (Ritchie and Thompson, 1974) and `S` (Becker and Chambers, 1984; Becker et al., 1988) have strongly influenced statistical software as we know it today. In fact, the `R` environment (R Core Team, 2014a), which features prominently in this thesis, is often dubbed the lingua franca of statistics and is a direct ancestor of the `S` language as originally proposed by John Chambers. In the traditions of Tukey and Bell Labs, the field has made remarkable progress over the past few decades. Specifically the `S` language has, to quote the Association for Computing Machinery, *"forever altered how people analyze, visualize, and manipulate data"* (ACM, 1998). Largely due to the availability of statistical software, data analysis has become the de facto method of empirical sciences, an integral part of academic curricula, and plays an increasingly prominent role in modern society.

However, half a century after Tukey first demonstrated how to compute the Fourier transform on a machine, big changes are on the horizon. Modern society has put data at the frontier for innovation and productivity, presenting problems of a unprecedented scale and complexity (Manyika et al., 2011). Developments in computer technology and scientific practices ask for more flexible and scalable analysis tools. At the same time, fast growing adjacent fields such as artificial intelligence, software engineering and graphic design are increasingly touching and overlapping with statistics. The combination and interaction of these factors is blurring the borders of traditional disciplines. Techniques for collection, management, and analysis of data have started to melt into a hybrid field of data. This joint discipline is enormous, which leads to the emergence of many new subdisciplines based on domain knowledge and technical expertise. A proliferation of short-lived ideas, paradigms, vocabulary, software, and business models has accompanied this tumultuous transition phase. The upcoming years will likely be decisive in determining where this transition converges, who takes the lead, and which branches are destined to become extinct.

It is in the interest of both the statistician and the scientific community that the statistical tradition and theoretical foundations do not get diluted in this transition. The experience and understanding of learning from data that exists in the literature and minds of the statistical community harbors unique value. Statistical principles are strongly rooted in probability theory and methods that have evolved over the years are best of breed and proven to be effective. Only statisticians truly master the fine art of carefully exploring and testing data in order to unveil relations, patterns, and stories hidden behind the numbers. This holistic approach of studying each individual dataset in all its facets is not very well understood by any other community. Moreover, the infrastructure and expertise in research and development of statistical methods will become only more valuable as the field expands. This wisdom and maturity provides the credentials for a leading role in the new era of data. Our discipline enjoys great respect from other fields and we are invited to show the way. However, we must also face our limitations. Modern data analysis involves technical challenges that are beyond our capabilities and require interdisciplinary collaboration. This changing reality asks for some reflection on what is the true strength of statistics and which tasks are perhaps better suited for other experts.

1.1.1 Size and scale

Amongst the most visible recent developments are those related to scale. The rapid growth of data and the rise of computationally intensive methods have multiplied demand for computational power. Over the past years, estimation based on resampling or Monte Carlo techniques has steadily gained popularity over analytical solutions. These new methods are often easier to develop and require less assumptions of the data, at the cost of computational energy. Therefore, many universities have invested in supercomputers that students and faculty share to schedule such computationally intensive processing jobs. More recently, a new class of problems under the umbrella term of *big data* has entered the arena. These are problems which by definition call for quantities of memory and cpu that a single machine can not possibly provide. Big data analysis needs to be spread out on computing clusters, which requires fundamental redesigns of analytical software to support distributed computing. Algorithms need to be based on naturally parallelizable design patterns such as *map-reduce* (Dean and Ghemawat, 2008) in order to generalize to large scale data processing.

It has become painfully clear that these challenges are beyond the domain of the traditional statistical software. The big data movement has been driven by the IT industry with a strong emphasis on software engineering solutions, such as distributed file-systems and tools for managing computing resources in clusters. Statistical products provide an interface to such systems at best. Big data software is still in its infancy and analysis programs that build on these stacks are quite rudimentary in comparison with mainstream statistical software. However, demand from governments and industries has made big data into big business, resulting in enormous efforts to close this gap. We are starting to see the first open source products that layer on Hadoop to implement natively distributed versions of statistical methods such as GLM, PCA and K-Means. Over the upcoming years, these systems will likely start replacing current software to provide better support for big data and distributed computing. Tools and techniques for data analysis originating from fields other than statistics are received with a mix of excitement, skepticism, and aversion that is dividing the community. However, it is absolutely critical to the survival of our discipline that statisticians join the big data movement and work with these new players to influence the next generation of analysis software.

1.1.2 Visualization

Alongside quantitative results such as averages and parameter estimates, another important device for inspecting data is provided by computer graphics. Statistical software packages typically include functionality to generate plots in some form or another. Implementations vary anywhere from low-level shape drawing to completely canned charts automatically included with particular analyses. For many years such graphing tools have provided a useful and popular complement to numerical methods in statistical software products. However visualization too has matured into its own field independent of statistical computing. Specialized programs and libraries have started to offer more powerful, flexible visualization capabilities than those found in statistical software, slowly taking away market share.

The definitive tipping point has been the introduction of advanced graphing capabilities `SVG`, `Canvas` and `WebGL` in web browsers as part of the `HTML5` standard in 2012. These technologies opened the door to high performance, vector based, interactive, and even `3D` visualization in web pages. They take advantage of dedicated languages for styling (`CSS`) and interaction (`JavaScript`) while leveraging a user base that literally includes the entire world. In very little time countless high quality open source `JavaScript` libraries have appeared that go far beyond traditional graphics. By taking advantage of internet access and the document object model (`DOM`), browser graphics introduce a new generation of visualizations including interactive maps, live data animation and data-driven documents (Bostock et al., 2011). But even for implementing traditional static plots, the browser offers unprecedented flexibility and performance and which eventually replace other graphics devices.

1.1.3 Domain specialization

The rise of data and statistics has given birth to many new specializations and subfields. Over the past two decades much of the social sciences, including psychology and political science have transitioned from mostly qualitative towards quantitative methods, making measurement and analysis of behavior the main subject of research. Other applied branches like biostatistics and econometrics have long outgrown their respective disciplines and are supporting dedicated degrees and departments. In bioinformatics, entire schools are forming around various types of genomic data, such as DNA microarrays and `ChIP`-sequencing. More recently, even areas like literature and journalism are discovering data as a source of information. These fields encounter data and problems of a different nature, but have already unveiled very promising applications. Following progress in academia, increasingly many organizations in the public and private sector are embracing statistical methods as well. Retailers rely on data analysis to predict sales, target advertisement, and improve recruiting, whereas governments use it to study demographics, evaluate policy, and mass surveillance to name a few applications.

An important consequence of these developments is that analysts and their tools become increasingly specialized. Whereas statistics used to be practiced mainly in universities and research labs, data analysis is now part of job descriptions in all corners of the economy. This transition shifts the emphasis of analysis tools away from general purpose software towards applications tailored specifically to data or problems as they appear in a particular field or occupation. To develop such tools, the statistician must work in a team of engineers, user interface designers and domain experts to implement custom applications with embedded analysis methods. This asks for a more flexible approach to statistical software which allows for incorporating domain knowledge and third party software in order to cater to the demands of specific user groups.

1.1.4 Socializing data analysis

A somewhat more cultural yet important trend in recent years has been the socializing of technology. Much of software innovation has shifted focus from optimizing ways in which users perform particular tasks towards improving communication and collaboration between people. One popular example is Github: a software hosting service with social networking features based on the revision control system `git` (Torvalds and Hamano, 2010). Its excellent branching and merging tools enable distributed non-linear development by making it effortless to fork, prototype, contribute, review, discuss, and incorporate code between users and projects. This system has revolutionized code management for open-source projects and has proven to catalyze efficiency and creativity in the development process. Many of the younger statistical software developers are already using Github or similar platforms to host and manage their scripts and packages.

There is a lot of room for further socializing the data analysis process itself as well, especially within the sciences. Practices in statistics have changed surprisingly little over the past few decades. Statisticians usually work with a locally installed software product to manipulate and analyze data, and then copy results into a report or presentation. Collaboration is often limited to sharing a script or data file on a personal homepage. Many have argued that modernizing these practices is long overdue and vital to the future of statistics. According to Lee et al. (2013) academic publication is on an irreversible trajectory moving in the direction of online, dynamic, open-access publishing models. Changing demands for reproducibility (Peng, 2011) and teaching (Nolan and Temple Lang, 2010) are also pushing for more transparent and accessible analysis tools. This coincides with the global movement of universities, governments and industries towards open data, open access knowledge and open source software. With the availability of enabling technologies, these developments suggest a future where reproducible data, code and results become an integral part of the scientific publication process. Internet based data analysis platforms with social networking features could greatly advance collaboration, peer review and education of statistics while at the same time addressing data management and scalability issues.

1.1.5 Integration and interoperability

In my opinion, all of these developments highlight the urgency of interdisciplinary collaboration and integration for the future of our field. Modern society presents problems that can only be tackled through joint effort of statisticians, engineers, computer scientists, web developers, graphic designers and domain experts. This requires statisticians to surrender some of their independence and learn to become better team players. Finding our new identity in this dynamic is perhaps the main challenge for the current generation of statisticians. However, doing everything ourselves is no longer feasible. Failure to adapt to this new role puts us at serious risk of becoming isolated and obsolete.

I am particularly concerned with the changing role of statistical software. It is my belief that *interoperability* of statistical tools with other software will be the key factor to sustained pertinence of the discipline. Today, statistical products are primarily designed as do-it-all standalone suites serving every need of the independent statistician. The technology and culture of statistical computing has been formed around the assumption that we are at the end of the software food chain. Statistical software packages make it easy to call out to a database, `C++` library or web `API`, but little thought and effort is put into interfacing statistical methods from third party software. Unfortunately this dynamic is not unlike the attitude of many statisticians themselves. In order to leverage statistical methods in systems, pipelines, and applications, the focus needs to shift away from standalone products for statisticians, towards modular analysis units. Rather than competing with big data software or browser based visualization, we should implement statistical methods that can integrate with these upcoming technologies. Tools that play nice with other software can facilitate better collaboration between statisticians and other team members and smoothen the transition into the interdisciplinary world. The demand for high quality analysis tools that the statistics community has to offer is enormous, but making them widely applicable requires some changes in the way we design our software. This brings us to the theme of this thesis: *embedded scientific computing.*

1.2 Motivation and scope

In 2009 I co-authored an article for the journal *Statistics in Medicine* (van Buuren and Ooms, 2009) which proposed a novel statistical method for diagnosing developmental disorders in young teenagers. Along with the paper and R implementation, we developed a free web service to directly use this method online without the need for technical knowledge or specialized software. The application was relatively primitive but succeeded in making a state of the art diagnosis tool widely available to clinics and hospitals in The Netherlands. It enabled physicians and pediatricians with limited experience in statistics or R to immediately take advantage of recent statistical innovation. For me, this was a proof of concept that showed the enormous potential of integrating open source analysis software in specialized applications. Scaling this up could greatly improve accessibility of statistical methods while multiplying return on investment of our efforts. In the period leading up to and during my graduate studies at UCLA, I have been involved in many similar projects developing software with embedded analysis and visualization, both in academic and commercial organizations. These included systems, web applications and reporting tools in areas such as health, education, geography, and finance. Experiences and struggles from such diverse projects provided unique insights in the recurring challenges of embedded scientific computing.

The topic of embedded scientific computing has not previously been studied with the attention and detail it deserves. The problems are underdetermined and their complexity is widely underestimated. Over the course of my studies I have come to realize that the true nature of the problems is not only caused by technical limitations, but has to be understood from a disconnect between disciplines in an interdisciplinary space. The subject intersects statistical computing, software engineering and applied data analysis, however insufficient overlap and joint work between researchers in these fields is impeding integration of tools and knowledge. Many technical problems arise due to lack of understanding of the practices in scientific computing and the type of problems that it encounters. Underappreciation of the domain logic and inner workings of statistical software has resulted in solutions that are impractical, unreliable or unscalable. Therefore, this dissertation starts with an in-depth exploration of what scientific computing is about and how it is different from other applications of computer science. By mapping out the domain logic and identifying important bottlenecks, I hope my work can contribute towards aligning the various efforts in this space and lead to a more coherent set of tools and knowledge for developing integrated analysis components.

The motivation and approach for this research were entirely bottom-up. The intention was never to solely apply or extend a particular existing technology or theoretical framework. Instead I studied the challenges and solutions as they arise in practice when implementing systems and applications with embedded analysis and visualization. From these experiences I tried to distill the principal problems in order to develop a general purpose software system. The process involves an empirical back and forth between diagnosing and refining problems, while experimenting with various approaches for solutions. This contrasts with most current research in statistics where implementation details are often considered an afterthought. In this thesis the software itself is the subject of the study and the conclusions largely follow from implementation rather than the other way around. Much effort has gone into developing high quality software to put various approaches to the test. The results are a convergence of many iterations of prototyping, testing, refactoring, and incorporating feedback.

1.2.1 Definition

The purpose of this research is to identify and discuss fundamental challenges of *scaling embedded scientific computing. Scientific computing* refers to the general class of computational methods for data manipulation, statistics, mining, algorithm development, analysis, visualization, and related functionality. Unfortunately there is no universally agreed-upon umbrella term for this genre of data-driven research software. Vocabulary and scope of the countless packages vary by domain and are constantly evolving. For example, `R` is officially a "software environment for statistical computing and graphics", but this does not exactly capture functionality for scraping, text processing, meta programming, and web development, to name some recent additions. Similarly, `Matlab` is marketed as a "numerical computing environment", `Julia` is "a programming language for technical computing", and `NumPy` is "the fundamental package for scientific computing with `Python`". In addition, many more specialized software packages use jargon specific to particular applications or domains. Each of these packages takes a unique angle, but they share a similar purpose and overlapping audience. Perhaps the most distinguishing characteristic is the emphasis of *data as objects* combined with a functional style of programming that resembles mathematical operations. No single description does justice to this continuously expanding body of methods and software. Within this thesis the various terms are used mostly interchangeably with minor differences in emphasis. But for the title *scientific computing* seems like the most general neutral term for this branch of computing.

The term *embedded* deserves some clarification as well. In the context of this thesis, it does not refer to software specifically written for a particular electronic or mechanical device, which is the conventional meaning of embedded systems in computer science. Instead, the term has been adopted from the `R` community. For example Neuwirth and Baier (2001) talked about "Embedding `R` in standard software" and Eddelbuettel and Francois (2011b) state that "The `RInside` package provides `C++` classes that make it easier to embed `R` in `C++` code". Moreover, the *Writing R Extensions* manual mentions the term frequently in the chapter "Embedding `R` under Unix-alikes" (R Core Team, 2014b). I use *embedded* to emphasize the role of software as a *component*, in contrast with a standalone application. In software engineering, a component is a module or web resource that encapsulates a set of related functions or data (Cox, 1990). It builds on the principle of separation of concerns and advocates a reuse-based approach to defining, implementing and composing loosely coupled independent modules into systems. Components are generally not interfaced by the user, but rather called from another piece of software through a programmable interface.

Finally, *scalability* connotes the ability of a system to accommodate an increasing number of elements or objects, to process growing volumes of work gracefully and be susceptible to enlargement (Bondi, 2000). Commercial vendors of statistical software often present scale exclusively in relation to the magnitude of the data and offer solutions to process more data in less time. Although important, there are many additional dimensions and directions to scale other than memory and disk space. For example, accommodating large amounts of users or concurrent tasks introduces new management and organization problems. Also systems that are high maintenance and require considerable human resources to accommodate growth do not scale well by definition. Furthermore, increased code complexity in large projects often reveals difficulties which were invisible on a smaller scale. Finally an important observation is that in order to scale up, systems must be able to scale down. Software which requires a special infrastructure and a team of experts and administrators to operate has limited applications, regardless of performance. On the other hand, a solution that has the potential to scale up to large problems, but also works for small projects by users with limited expertise is much more likely to find wide adoption. Human dimensions to scalability are easily overlooked, but in practice just as critical as overcoming technical limitations.

1.2.2 Overview

The main body of the thesis consists of four contributions which form the corner stones of the research. These pieces treat the most pressing and difficult problems that are encountered when building systems with embedded scientific computing. Chapter 2 presents the central work of this thesis and treats the problem of interfacing computational procedures. Currently most statistical software is designed for `UI` rather than `API` interaction. However, clients or systems integrating analytical components require a programmable interface that exposes desired functionality while abstracting implementation details. The principle of separation of concerns requires us to carefully carve out logic specific to scientific computing and distinguish it from any application or implementation logic. To this end, the majority of the chapter is concerned with a high level description of the principles and practices of scientific computing. The `OpenCPU` system is introduced as a proof of concept and reference implementation that puts the advocated approach in practice. Experiences from developing and applying the `OpenCPU` software are a driving factor throughout the research and also motivate the remaining chapters of this thesis.

The subsequent chapters treat more technical issues that come up with embedded scientific computing. Chapter 3 talks about enforcing security policies and managing hardware resources. Because statistical software products are traditionally designed for local use, access control is generally not considered an issue and the execution environment is entirely unrestricted. However, embedded statistical components require fine grained control over permissions and resources allocated to individual users or tasks. The chapter discusses various security models and explains the domain specific aspects of the problem that make it difficult to apply general purpose solutions. The `RAppArmor` package is introduced to illustrate *mandatory access control* style security in `R` on `Linux` systems. This allows for divorcing the security concern from the application layer and solve it on the level of the operating system. `RAppArmor` is at the basis of the `OpenCPU` cloud server implementation and makes it possible to expose arbitrary code execution privileges to the public. This provides a basis for applications where users can freely share, modify and run custom code which is central to the `OpenCPU` philosophy.

Chapter 4 discusses the issue of data interchange, specifically in the context of the JSON format. A key challenge for interfacing statistical components is getting data in and out of the system. To this end, OpenCPU supports two popular data interchange formats: JSON and Protocol Buffers (Eddelbuettel et al.). However the main difficulties are not so much related to the format, but rather to the *structure* of input and output data. Strong typed languages typically use schemas to formally describe and enforce structure. However this is not very natural for dynamically typed languages with loosely defined data structures frequently used in scientific computing. Instead we take an approach that directly maps the most important data types in R to JSON and vice versa. This allows clients to interact with R functions and objects via JSON without any specific knowledge about implementation of object classes in R. The jsonlite package is used as a reference implementation throughout the chapter to illustrate this mapping. Besides OpenCPU, at least a dozen other projects have already adopted the jsonlite package to interact with JSON data in R as well.

Finally, chapter 5 treats the more high level issue of managing software versioning and specifically inter-software dependencies in the context of scientific computing. In my experience, imprudent software versioning is by far the primary cause of problems in statistical software. Rapidly growing repositories have made current practices unsustainable and properly addressing this issue is critical for turning statistical methods into reliable components and applications. Unfortunately the problem is not sufficiently understood and acknowledged by most members of the community. This can be traced back to a long culture of exclusively interactive use which assumes that the user will manually manage software and debug problems as they appear. The chapter explores the problem in great detail and explains how limitations of dependency versioning are causing unstable software and irreproducible results. I make a case to the R community for moving to a design that better accommodates modern software development and can scale to large repositories.

1.2.3 About R

This research relies heavily on the *The R Project for Statistical Computing*, for short: `R` (R Core Team, 2014a). The `R` language is currently the most popular open source software for statistical computing and considered by many statisticians as the de facto standard for data analysis. The software is well established and the huge `R` community provides a wealth of contributed packages in a variety of fields. Currently only `R` has the required maturity and infrastructure to build and test scalable computing systems. Therefore, `R` is the obvious candidate to experiment with embedded scientific computing. Most of my experiences that led to this research have been with `R`, and the software presented in this thesis is implemented on `R`. However, I want to emphasize that the purpose of this research was not just to develop software and the problems discussed are not limited to any particular implementation. Difficulties related to interfacing, security, data interchange and dependency management will appear in some form or another when implementing analysis components in any language. None of the currently available systems, including commercial products, provide any simple and general solutions.

Yet in reality it is very difficult to study or discuss software without an actual implementation. Only by developing and using software we discover the strengths and limitations of certain technology and learn which solutions are practical and reliable. For these reasons, the problems of embedded scientific computing are mostly treated from the way they take shape in `R`. Rather than presenting abstract ideas in general terms, we take advantage of the lingua franca to demonstrate problems and solutions as they appear in practice. Therefore each chapter discusses some relevant aspects of the `R` language that provide the context for a particular issue. For example, the conventional container format for namespacing a collection of data and functions in `R` is a *package*. Even though other products might use different mechanics and jargon for combining and publishing a set of objects, any system needs some notion of namespaces and dependencies analogous to packages in `R`. Throughout the thesis I discuss concepts using terminology and examples from the `R` language, but repeatedly emphasize that software serves primarily as illustration. Similar techniques used in the `R` packages should work when building analysis components with `Julia`, `Matlab` or `Python`.

CHAPTER 2

The OpenCPU System: Towards a Universal Interface for Scientific Computing through Separation of Concerns

2.1 Introduction

Methods for scientific computing are traditionally implemented in specialized software packages assisting the statistician in all facets of the data analysis process. A single product typically includes a wealth of functionality to interactively manage, explore and analyze data, and often much more. Products such as `R` or `STATA` are optimized for use via a command line interface (`CLI`) whereas others such as `SPSS` focus mainly on the graphical user interface (`GUI`). However, increasingly many users and organizations wish to integrate statistical computing into third party software. Rather than working in a specialized statistical environment, methods to analyze and visualize data get incorporated into pipelines, web applications and big data infrastructures. This way of doing data analysis requires a different approach to statistical software which emphasizes interoperability and programmable interfaces rather than `UI` interaction. Throughout the paper we refer to this approach to statistical software as *embedded scientific computing.*

Previous work in embedded scientific computing has mostly been limited to low-level tools for connecting statistical software to general purpose environments. For `R`, bindings and bridges are available to execute an `R` script or process from inside all popular languages. For example, JRI (Urbanek, 2013b), RInside (Eddelbuettel and Francois, 2011b), rpy2 (Gautier, 2012) or RinRuby (Dahl and Crawford, 2009) can be used to call R from respectively Java, C++, Python or Ruby. Heiberger and Neuwirth (2009) provide a set of tools to run R code from DCOM clients on Windows, mostly to support calling R in Microsoft Excel. The rApache module (mod_R) makes it possible to execute R scripts from the Apache2 web server (Horner, 2013). Similarly, the littler program provides hash-bang capability for R, as well as simple command-line and piping use on UNIX (Horner and Eddelbuettel, 2011). Finally, Rserve is TCP/IP server which provides low level access to an R process over a socket (Urbanek, 2013d).

Even though these bridging tools have been available for several years, they have not been able to facilitate the big breakthrough of R as a ubiquitous statistical engine. Given the enormous demand for analysis and visualization these days, the adoption of R for embedded scientific computing is actually quite underwhelming. In my experience, the primary cause for the limited success is that these bridges are hard to implement, do not scale very well, and leave the most challenging problems unresolved. Substantial plumbing and expertise of R internals is required for building actual applications on these tools. Clients are supposed to generate and push R syntax, make sense of R's internal C structures and write their own framework for managing requests, graphics, security, data interchange, etc. Thereby, scientific computing gets intermingled with other parts of the system resulting in highly coupled software which is complex and often unreliable. High coupling is also problematic from a human point of view. Building a web application with for example Rserve requires a web developer that is also an expert in R, Java and Rserve. Because R is a very domain specific language, this combination of skills is very rare and expensive.

2.1.1 Separation of concerns

What is needed to scale up embedded scientific computing is a system that decouples data analysis from other system components in such a way that applications can integrate statistical methods without detailed understanding of `R` or statistics. Component based software engineering advocates the design principle of *separation of concerns* (Heineman and Councill, 2001), which states that a computer program is split up into distinct pieces that each encapsulate a *logical* set of functionality behind a well-defined interface. This allows for independent development of various components by different people with different background and expertise. Separation of concerns is fundamental to the functional programming paradigm (Reade, 1989) as well as the design of service oriented architectures on distributed information systems such as the internet (Fielding, 2000). The principle lies at the heart of this research and holds the key to advancing embedded scientific computing.

In order to develop a system that separates concerns of scientific computing from other parts of the system, we need to ask ourselves: what are the concerns of scientific computing? This question does not have a straightforward answer. Over the years, statistical software has gotten highly convoluted by the inclusion of complementary tools that are useful but not necessarily an integral part of computing. Separation of concerns requires us to extract the core logic and divorce it from all other apparatus. We need to form a conceptual model of data analysis that is independent of any particular application or implementation. Therefore, rather than discussing technical problems, this paper focuses entirely on studying the domain logic of the discipline along the lines of Evans (2003). By exploring the concepts, problems, and practices of the field we try to unveil the fundamental principles behind statistical software. Along the way we highlight important problems and bottlenecks that require further attention in order to facilitate reliable and scalable analysis modules.

The end goal of this paper is to work towards an interface definition for embedded scientific computing. An interface is the embodiment of separation of concerns and serves as a contract that formalizes the boundary across which separate components exchange information. The interface definition describes the concepts and operations that components agree upon to cooperate and how the communication is arranged. In the interface we specify the functionality that a server has to implement, which parts of the interaction are fixed and which choices are specifically left at the discretion of the implementation. Ideally the specification should provide sufficient structure to develop clients and server components for scientific computing while minimizing limitations on how these can be implemented. An interface that carefully isolates components along the lines of domain logic allows developers to focus on their expertise using their tools of choice. It gives clients a universal point of interaction to integrate statistical programs without understanding the actual computing, and allows statisticians to implement their methods for use in applications without knowing specifics about the application layer.

2.1.2 The OpenCPU system

The `OpenCPU` system is an example that illustrates what an abstracted interface to scientific computing could look like. `OpenCPU` defines an `HTTP API` that builds on *The R Project for Statistical Computing*, for short: `R` (R Core Team, 2014a). The `R` language is the obvious candidate for a first implementation of this kind. It is currently the most popular statistical software package and considered by many statisticians as the de facto standard of data analysis. The huge `R` community provides both the tools and use-cases needed to develop and experiment with this new approach to scientific computing. It is fair to say that currently only `R` has the required scale and foundations to really put our ideas to the test. However, although the research and `OpenCPU` system are colored by and tailored to the way things work in R, the approach should generalize quite naturally to other computational back-ends. The API is designed to describe general logic of data analysis rather than that of a particular language. The main role of the software is to put this new approach into practice and get firsthand experience with the problems and opportunities in this unexplored field.

As part of the research, two OpenCPU server implementations were developed. The R package opencpu uses the httpuv web server (RStudio Inc., 2014a) to implement a *single-user server* which runs within an interactive R session on any platform. The *cloud server* on the other hand is a multi-user implementation based on Ubuntu Linux and rApache. The latter yields much better performance and has advanced security and configuration options, but requires a dedicated Linux server. Another major difference between these implementations is how they handle concurrency. Because R is single threaded, httpuv handles only a single request at a time. Additional incoming requests are automatically queued and executed in succession using the same process. The cloud server on the other hand takes advantage of multi-processing in the Apache2 web server to handle concurrency. This implementation uses forks of the R process to serve concurrent requests immediately with little performance overhead. The differences between the cloud server and single user server are invisible the client. The API provides a standard interface to either implementation and other than varying performance, applications will behave the same regardless of which server is used. This already hints at the benefits of a well defined interface.

2.1.3 History of OpenCPU

The OpenCPU system builds on several years of work dating back to 2009. The software evolved through many iterations of trial and error by which we identified the main concerns and learned what works in practice. Initial inspirations were drawn from recurring problems in developing R web applications with rApache, including van Buuren and Ooms (2009). Accumulated experiences from these projects shaped a vision on what is involved with embedded scientific computing. After a year of internal development, the first public beta of `OpenCPU` appeared in August 2011. This version was picked up by early adopters in both industry and academia, some of which are still in production today. The problems and suggestions generated from early versions were a great source of feedback and revealed some fundamental problems. At the same time exciting developments were going on in the `R` community, in particular the rise of the `RStudio IDE` and introduction of powerful new `R` packages `knitr`, `evaluate` and `httpuv`. After a redesign of the `API` and a complete rewrite of the code, `OpenCPU 1.0` was released in August 2013. By making better use of native features in `HTTP`, this version is more simple, flexible, and extensible than before. Subsequent releases within the `1.x` series have introduced additional server configurations and optimizations without major changes to the `API`.

2.2 Practices and domain logic of scientific computing

This section provides a helicopter view of the practices and logic of scientific computing that are most relevant in the context of this research. The reader should get a sense of what is involved with scientific computing, what makes data analysis unique, and why the software landscape is dominated by domain specific languages (`DSL`). The topics are chosen and presented somewhat subjectively based on my experiences in this field. They are not intended to be exhaustive or exclusive, but rather identify the most important concerns for developing embedded analysis components.

2.2.1 It starts with data

The role and shape of *data* is the main characteristic that distinguishes scientific computing. In most general purpose programming languages, *data structures* are instances of classes with well-defined fields and methods. Similarly, databases use schemas or table definitions to enforce the structure of data. This ensures that a table returned by a given `SQL` query always contains exactly the same structure with the requested fields; the only varying property between several executions of a query is the number of returned rows. Also the time needed for the database to process the request usually depends only on the amount of records in the database. Strictly defined structures make it possible to write code implementing all required operations in advance without knowing the actual *content* of the data. It also creates a clear separation between developers and users. Most applications do not give users direct access to raw data. Developers focus in implementing code and designing data structures, whereas users merely get to execute a limited set of operations.

This paradigm does not work for scientific computing. Developers of statistical software have relatively little control over the structure, content, and quality of the data. Data analysis starts with the user supplying a dataset, which is rarely pretty. Real world data come in all shapes and formats. They are messy, have inconsistent structures, and invisible numeric properties. Therefore statistical programming languages define data structures relatively loosely and instead implement a rich lexicon for interactively manipulating and testing the data. Unlike software operating on well-defined data structures, it is nearly impossible to write code that accounts for any scenario and will work for every possible dataset. Many functions are not applicable to every instance of a particular class, or might behave differently based on dynamic properties such as size or dimensionality. For these reasons there is also less clear of a separation between developers and users in scientific computing. The data analysis process involves simultaneously debugging of code and data where the user iterates back and forth between manipulating and analyzing the data. Implementations of statistical methods tend to be very flexible with many parameters and settings to specify behavior for the broad range of possible data. And still the user might have to go through many steps of cleaning and reshaping to give data the appropriate structure and properties to perform a particular analysis.

Informal operations and loosely defined data structures are typical characteristics of scientific computing. They give a lot of freedom to implement powerful and flexible tools for data analysis, but complicate interfacing of statistical methods. Embedded systems require a degree of type-safety, predictability, and consistency to facilitate reliable `I/O` between components. These features are native to databases or many object oriented languages, but require substantial effort for statistical software.

2.2.2 Functional programming

Many different programming languages and styles exists, each with their own strengths and limitations. Scientific computing languages typically use a functional style of programming, where methods take a role and notation similar to *functions* in mathematics. This has obvious benefits for numerical computing. Because equations are typically written as $y = f(g(x))$ (rather than $y = x.g().f()$ notation), a functional syntax results in intuitive code for implementing algorithms.

Most popular general purpose languages take a more imperative and object oriented approach. In many ways, object-oriented programming can be considered the opposite of functional programming (A. M. Kuchling, 2014). Here methods are invoked on an object and modify the *state* of this particular object. Object-oriented languages typically implement inheritance of fields and methods based on object classes or prototypes. Many software engineers prefer this style of programming because it is more powerful to handle complex data structures. The success of object oriented languages has also influenced scientific computing, resulting in multi-paradigm systems. Languages such as `Julia` and `R` use multiple dispatch to dynamically assign function calls to a particular function based on the type of arguments. This brings certain object oriented benefits to functional languages, but also complicates scoping and inheritance.

A comparative review on programming styles is beyond the scope of this research. But what is relevant to us is how conflicting paradigms affect interfacing of analysis components. In the context of web services, the *Representational State Transfer* style (for short: `REST`) described by Fielding (2000) is very popular among web developers. A *restful* `API` maps every `URL` to a *resource* and `HTTP` requests are used to modify the *state* of a resource, which results in a simple and elegant `API`. Unfortunately, `REST` does not map very naturally to the functional paradigm of statistical software. Languages where functions are first class citizens suggest more `RPC` flavored interfaces, which according to Fielding are by definition not restful (Fielding, 2008). This does not mean that such a component is incompatible with other pieces. As long as components honor the rules of the protocol (i.e. `HTTP`) they will work together. However, conflicting programming styles can be a source of friction for embedded scientific computing. Strongly object-oriented frameworks or developers might require some additional effort to get comfortable with components implementing a more functional paradigm.

2.2.3 Graphics

Another somewhat domain specific feature of scientific computing is native support for graphics. Most statistical software packages include programs to draw plots and charts in some form or another. In contrast to data and functions which are language objects, the graphics device is considered a separate output stream.

Drawings on the canvas are implemented as a side effect rather than a return value of function calls. This works a bit similar to manipulating document object model (`DOM`) elements in a browser using `JavaScript`. In most interactive statistical software, graphics appear to the user in a new window. The state of the graphics device cannot easily be stored or serialized as is the case for functions and objects. We can export an *image* of the graphics device to a file using `png`, `svg` or `pdf` format, but this image is merely a snapshot. It does not contain the actual state of the device cannot be reloaded for further manipulation.

First class citizenship of graphics is an important concern of interfacing scientific computing. Yet output containing both data and graphics makes the design of a general purpose `API` more difficult. The system needs to capture the return value as well as graphical side effects of a remote function call. Furthermore the interface should allow for generating graphics without imposing restrictions on the format or formatting parameters. Users want to utilize a simple bitmap format such as `png` for previewing a graphic, but have the option to export the same graphic to a high quality vector based format such as `pdf` for publication. Because statistical computation is expensive and non-deterministic, the graphic cannot simply reconstructed from scratch only to retrieve it in another format. Hence the `API` needs to incorporate the notion of a graphics device in a way independent of the imaging format.

2.2.4 Numeric properties and missing values

It was already mentioned how loosely defined data structures in scientific computing can impede type safety of data `I/O` in analysis components. In addition, statistical methods can choke on the actual content of data as well. Sometimes problematic data can easily be spotted, but often it is nearly impossible to detect these ahead of time. Applying statistical procedures to these data will then result in errors, even though the code and structure of the data are perfectly fine. These problems frequently arise for statistical models that build on matrix decompositions which require the data to follow certain numeric properties. The *rank* of a matrix is one such property which measures the nondegenerateness of the system of linear equations. When a matrix A is rank deficient, the equation $Ax = b$ does not have a solution when b does not lie in the range of A. Attempting to solve this equation will eventually lead to division by zero. Accounting for such cases of time is nearly impossible because numeric properties are invisible until they are actually calculated. But perhaps just as difficult is explaining the user or software engineer that these errors are not a bug, and can not be fixed. The procedure just does not work for this particular dataset.

Another case of problematic data is presented by *missing* values. Missingness in statistics means that the value of a field is unknown. Missing data should not be confused with no data or `null`. Missing values are often *non ignorable*, meaning that the missingness itself is information that needs to be accounted for in the modeling. A standard textbook example is when we perform a survey asking people about their salary. Because some people might refuse to provide this information, the data contains missing values. This missingness is probably *not completely at random*: respondents with high salaries might be more reluctant to provide this information than respondents with a median salary. If we calculate the mean salary from our data ignoring the missing values, the estimate is likely biased. To obtain a more accurate estimate of the average salary, missing values need to be incorporated in the estimation using a more sophisticated model.

Statistical programming languages can define several types of missing or non-finite values such as `NA`, `NaN` or `Inf`. These are usually implemented as special primitives, which is one of the benefits of using a `DSL`. Functions in statistical software have built-in procedures and options to specify how to handle missing values encountered in the data. However, the notion of missingness is foreign to most languages and software outside of scientific computing. They are a typical domain-specific phenomenon that can cause technical problems in data exchange with other systems. And like numeric properties, the concept of values containing no actual value is likely to cause confusion among developers or users with limited experience in data analysis. Yet failure to properly incorporate missing values in the data can easily lead to errors or incorrect results, as the example above illustrated.

2.2.5 Non deterministic and unpredictable behavior

Most software applications are expected to produce consistent output in a timely manner, unless something is very wrong. This does not generally hold for scientific computing. The previous section explained how problematic data can cause exceptions or unexpected results. But many analysis methods are actually non-deterministic or unpredictable by nature.

Statistical algorithms often repeat some calculation until a particular convergence criterion is reached. Starting values and minor fluctuations in the data can have snowball effect on the course of the algorithm. Therefore several runs can result in wildly varying outcomes and completion times. Moreover, convergence might not be guaranteed: unfortunate input can get a process stuck in a local minimum or send it off into the wrong direction. Predicting and controlling for such scenarios a-priori in the implementation is very difficult. Monte Carlo techniques are even less predictable because they are specifically designed to behave randomly. For example, `MCMC` methods use a Markov-Chain to simulate random walk through a (high-dimensional) space such as a multivariate probability density. These methods are a powerful tool for simulation studies and numerical integration in Bayesian analysis. Each execution of the random walk yields different outcomes, but under general conditions the process will converge to the value of interest. However, due to randomness it is possible that some of the runs or chains get stuck and need to be terminated or disregarded.

Unpredictability of statistical methods underlies many technical problems for embedded scientific computing that are not present when interacting with a database. This can sometimes surprise software engineers expecting deterministic behavior. Statistical methods are rarely absolutely guaranteed to be successful for arbitrary data. Assuming that a procedure will always return timely and consistently because it did so with testing data is very dangerous. In a console, the user can easily intervene or recover, and retry with different options or starting values. For embedded modules, unpredictability needs to be accounted for in the design of the system. At a very minimum, the system should be able to detect and terminate a process that has not completed when some timeout is reached. But preferably we need a layer or meta functionality to control and monitor executions, either manually or automatically.

2.2.6 Managing experimental software

In scientific computing, we usually need to work with inventive, volatile, and experimental software. This is a big cultural difference with many general purpose languages such as `python`, `Java`, `C++` or `JavaScript`. The latter communities include professional organizations and engineers committed to implementing and maintaining production quality libraries. Most authors of open source statistical software do not have the expertise and resources to meet such standards. Contributed code in languages like `R` was often written by academics or students to accompany a scientific article proposing novel models, algorithms, or programming techniques. The script or package serves as an illustration of the presented ideas, but needs needs to be tweaked and tailored to fit a particular problem or dataset. The quality of such contributions varies a lot, no active support or maintenance should be expected from the authors. Furthermore, package updates can sometimes introduce radical changes based on new insights.

Because traditional data analysis does not really have a notion of production, this has never been a major problem. The emphasis in statistical software has always been on innovation rather than continuity. Experimental code is usually good enough for interactive data analysis where it suffices to manually make a script or package work for the dataset at hand. Authors of statistical software tend to assume that the user will spend some effort to manage dependencies and debug the code. However, integrated components require a greater degree of reliability and continuity which introduces a source of technical and cultural friction for embedded scientific computing. This makes the ability to manage unstable software, facilitate rapid change, sandbox modules, and manage failure important concerns of embedded scientific computing.

2.2.7 Interactivity and error handling

In general purpose languages, run-time errors are typically caused by a bug or some sort of system failure. Exceptions are only raised when the software can not recover and usually result in termination of the process. Error messages contain information such as calling stacks to help the programmer discover where in the code a problem occurred. Software engineers go through great trouble to prevent potential problems ahead of time using smart compilers, unit tests, automatic code analysis, and continuous integration. Errors that do arise during production are usually not displayed to the user, but rather the administrator is notified that the system urgently needs attention. The user gets to see an apology at best.

In scientific computing, errors play a very different role. As a consequence of some of the characteristics discussed earlier, interactive debugging is a natural part of the user experience. Errors in statistics do not necessarily indicate a bug in the software, but rather a problem with the data or some interaction of the code and data. The statistician goes back and forth between cleaning, manipulating, modeling, visualizing and interpreting to study patterns and relations in the data. This simultaneous debugging of data and code comes down to a lot of trial and error. Problems with outliers, degrees of freedom or numeric properties do not reveal themselves until we try to fit a model or create a plot. Exceptions raised by statistical methods are often a sign that data needs additional work. This makes error messages an important source of information for the statistician to get to know a dataset and its intricacies. And while debugging the data we learn limitations of the analysis methods. In practice we sometimes find out that a particular dataset requires us to research or implement additional techniques because the standard tools do not suffice or are inappropriate.

Interactive error handling is one of the reasons that there is no clear distinction between development and production in scientific computing. When interfacing with analysis modules it is important that the role of errors is recognized. An `API` must be able to handle exceptions and report error messages to the user, and certainly not crash the system. The role of errors and interactive debugging in data analysis can be confusing to developers outside of our community. Some popular commercial products seem to have propagated the belief that data analysis comes down to applying a magical formula to a dataset, and no intelligent action is required on the side of the user. Systems that only support such canned analyses don't do justice to the wide range of methods that statistics has to offer. In practice, interactive data debugging is an important concern of data analysis and embedded scientific computing.

2.2.8 Security and resource control

Somewhat related to the above are special needs in terms of security. Most statistical software currently available is primarily designed for interactive use on the local machine. Therefore access control is not considered an issue and the execution environment is entirely unrestricted. Embedded modules or public services require implementation of security policies to prevent malicious or excessive use of resources. This in itself is not a unique problem. Most scripting languages such as `php` or `python` do not enforce any access control and assume security will be implemented on the application level. But in the case of scientific computing, two domain specific aspects further complicate the problem.

The first issue is that statistical software can be demanding and greedy with hardware resources. Numerical methods are expensive both in terms of memory and cpu. Fair-use policies are not really feasible because excessive use of resources often happens unintentionally. For example, an overly complex model specification or algorithm getting stuck could end up consuming all available memory and cpu until manually terminated. When this happens on the local machine, the user can easily interrupt the process prematurely by sending a `SIGINT` (pressing `CTRL+C` or `ESC`), but in a shared environment this needs to be regulated by the system. Embedded scientific computing requires technology and policies that can manage and limit memory allocation, cycles, disk space, concurrent processes, network traffic, etc. The degree of flexibility offered by implementation of resource management is an important factor in the scalability of a system. Fine grained control over system resources consumed by individual tasks allows for serving many users without sacrificing reliability.

The second domain specific security issue is caused by the need for arbitrary code execution. A traditional application security model is based on user role privileges. In a standard web application, only a developer or administrator can implement and deploy actual code. The application merely exposes predefined functionality; users are not allowed to execute arbitrary code on the server. Any possibility of code injection is considered a security vulnerability and when found the server is potentially compromised. However as already mentioned, lack of segregation between users and developers in statistics gives limited use to applications that restrict users to predefined scripts and canned services. To support actual data analysis, the user needs access to the full language lexicon to freely explore and manipulate the data. The need for arbitrary code execution disqualifies user role based privileges and demands a more sophisticated security model.

2.2.9 Reproducible research

Replication of findings is one of the main principles of the scientific method. In quantitative research, a necessary condition for replication is reproducibility of results. The goal of reproducible research is to tie specific instructions to data analysis and experimental data so that scholarship can be recreated, better understood, and verified (Kuhn, 2014). Even though the ideas of replication are as old as science itself, reproducibility in scientific computing is still in its infancy. Tools are available that assist users in documenting their actions, but to most researchers these are not a natural part of the daily workflow. Fortunately, the importance of replication in data analysis is increasingly acknowledged and support for reproducibility is becoming more influential in the design of statistical software.

Reproducibility changes what constitutes the main product of data analysis. Rather than solely output and conclusions, we are interested recording and publishing the entire *analysis process*. This includes all data, code and results but also external software that was used arrive at the results. Reproducibility puts high requirements on software versioning. More than in other fields it is crucial that we diligently archive and administer the precise versions or branches of all scripts, packages, libraries, plugins that were somehow involved in the process. If an analysis involves randomness, it is also important that we keep track of which seeds and random number generators were used. In the current design of statistical software, reproducibility was mostly an afterthought and has to be taken care of manually. In practice it is tedious and error-prone. There is a lot of room for improvement through software that incorporates reproducible practices as a natural part of the data analysis process.

Whereas reproducibility in statistics is acknowledged from a transparency and accountability point of view, it has enormous potential to become much more than that. There are interesting parallels between reproducible research and revision control in source code management systems. Technology for automatic reproducible data analysis could revolutionize scientific collaboration, similar to what `git` has done for software development. A system that keeps track of each step in the analysis process like a *commit* in software versioning would make peer review or follow-up analysis more practical and enjoyable. When colleagues or reviewers can easily reproduce results, test alternative hypotheses or recycle data, we achieve greater trustworthiness but also multiply return on investment of our work. Finally an open kitchen can help facilitate more natural ways of learning and teaching statistics. Rather than relying on general purpose textbooks with artificial examples, scholars directly study the practices of prominent researchers to understand how methods are applied in the context of data and problems as they appear specifically in their area of interest.

2.3 The state problem

Management of *state* is a fundamental principle around which digital communications are designed. We distinguish *stateful* and *stateless* communication. In a stateless communication protocol, interaction involves independent request-response messages in which each request is unrelated by any previous request (Hennessy and Patterson, 2011). Because the messages are independent, there is no particular ordering to them and requests can be performed concurrently. Examples of stateless protocols include the internet protocol (`IP`) and the hypertext transfer protocol (`HTTP`). A stateful protocol on the other hand consists of an interaction via an ordered sequence of interrelated messages. The specification typically prescribes a specific mechanism for initiating and terminating a persistent *connection* for information exchange. Examples of stateful protocols include the transmission control protocol (`TCP`) or file transfer protocol (`FTP`).

In most data analysis software, the user controls an interactive session through a console or `GUI`, with the possibility of executing a sequence of operations in the form of a *script*. Scripts are useful for publishing code, but the most powerful way of using the software is interactively. In this respect, statistical software is not unlike to a shell interface to the operating system. Interactivity in scientific computing makes management of state the most central challenge in the interface design. When moving from a `UI` to `API` perspective, support for statefulness becomes substantially more complicated. This section discusses how the existing bridges to `R` have approached this problem, and their limitations. We then continue by explaining how the `OpenCPU API` exploits the functional paradigm to implement a hybrid solution that abstracts the notion of state and allows for a high degree of performance optimization.

2.3.1 Stateless solutions: predefined scripts

The easiest solution is to not incorporate state on the level of the interface, and limit the system to predefined scripts. This is the standard approach in traditional web development. The web server exposes a parameterized service which generates dynamic content by calling out to a script on the system via `CGI`. Any support for state has to be implemented manually in the application layer, e.g. by writing code that stores values in a database. For `R` we can use `rApache` (Horner, 2013) to develop this kind of scripted applications very similar to web scripting languages such as `php`. This suffices for relatively simple services that expose limited, predefined functionality. Scripted solutions give the developer flexibility to freely define input and output that are needed for a particular application. For example, we can write a script that generates a plot based on a couple of input parameters and returns a fixed size `png` image. Because scripts are stateless, multiple requests can be performed concurrently. A lot of the early work in this research has been based on this approach, which is a nice starting point but becomes increasingly problematic for more sophisticated applications.

The main limitation of scripts is that to support basic interactivity, retention of state needs to be implemented manually in the application layer. A minimal application in statistics consists of the user uploading a data file, performing some manipulations and then creating a model, plot or report. When using scripts, the application developer needs to implement a framework to manage requests from various user sessions, and store intermediate results in a database or disk. Due to the complexity of objects and data in `R`, this is much more involved than it is in e.g. `php`, and requires programming skills. Furthermore it leads to code that intermingles scientific computing with application logic, and rapidly increases complexity as the application gets extended with additional scripts. Because these problems will recur for almost any statistical application, we could benefit greatly from a system that supports retaining state by design.

Moreover predefined scripts are problematic because they divide developers and users in a way that is not very natural for scientific computing. Scripts in traditional web development give the client very little power to prevents malicious use of services. However, in scientific computing, a script often merely serves as a starting point for analysis. The user wants to be able to modify the script to look at the data in another way by trying additional methods or different procedures. A system that only allows for performing scripted actions severely handicaps the client and creates a lot of work for developers: because all functionality has to be prescripted, they are in charge of designing and implementing each possible action the user might want to perform. This is impractical for statistics because of the infinite amount of operations that can be performed on a dataset. For these reasons, the stateless scripting approach does not scale well to many users or complex applications.

2.3.2 Stateful solution: client side process management

Most existing bridges to `R` have taken a stateful approach. Tools such as `Rserve` (Urbanek, 2013d) and `shiny` (RStudio Inc., 2014b) expose a low-level interface to a private `R` process over a (web)socket. This gives clients freedom to run arbitrary `R` code, which is great for implementing something like a web-based console or `IDE`. The main problem with existing stateful solutions is lack of interoperability. Because these tools are in essence a remote `R` console, they do not specify any standardized interface for calling methods, data `I/O`, etc. A low-level interface requires extensive knowledge of logic and internals of `R` to communicate, which again leads to high coupling. The client needs to be aware of `R` syntax to call `R` methods, interpret `R` data structures, capture graphics, etc. These bridges are typically intended to be used in combination with a special client. In the case of `shiny`, the server comes with a set of widget templates that can be customized from within `R`. This allows `R` users to create a basic web `GUI` without writing any `HTML` or `JavaScript`, which can be very useful. However, the shiny software is not designed for integration with non-shiny clients and serves a somewhat different purpose and audience than tools for embedded scientific computing.

Besides high coupling and lack of interoperability, stateful bridges also introduce some technical difficulties. Systems that allocate a private `R` process for each client cannot support concurrent requests within a session. Each incoming request has to wait until the previous requests are finished for the process to become available. In addition to suboptimal performance, this can also be a source of instability. When the `R` process gets stuck or raises an unexpected error, the server might become unresponsive causing the application to crash. Another drawback is that stateful servers are extremely expensive and inefficient in terms of memory allocation. The server has to keep each `R` process alive for the full duration of a session, even when idle most of the time. Memory that is in use by any single client does not free up until the user closes the application. This is particularly unfortunate because memory is usually the main bottleneck in data intensive applications of scientific computing. Moreover, connectivity problems or ill-behaved clients require mechanisms to timeout and terminate inactive processes, or save and restore an entire session.

2.3.3 A hybrid solution: functional state

We can take the best of both worlds by abstracting the notion of state to a higher level. Interactivity and state in `OpenCPU` is provided through persistence of *objects* rather than a persistent *process*. As it turns out, this is a natural and powerful definition of state within the functional paradigm. Functional programming emphasizes that output from methods depends only on their inputs and not on the program state. Therefore, functional languages can support state without keeping an entire process alive: merely retaining the state of objects should be sufficient. As was discussed before, this has obvious parallels with mathematics, but also maps beautifully to stateless protocols such as `HTTP`. The notion of state as the set of objects is already quite natural to the `R` user, as is apparent from the `save.image` function. This function serializes all objects in the global environment to a file on disk which described in the documentation as "saving the current workspace". Exploiting this same notion of state in our interface allows us to get the benefits of both traditional stateless and stateful approaches without introducing additional complexity. This simple observation provides the basis for a very flexible, stateful `RPC` system.

To facilitate this, the `OpenCPU API` defines a mapping between `HTTP` requests and `R` function calls. After executing a function call, the server stores all outputs (return value, graphics, files) and a *temporary key* is given to the client. This key can be used to control these newly created resources in future requests. The client can retrieve objects and graphics in various formats, publish resources, or use them as arguments in subsequent function calls. An interactive application consists of a series of RPC requests with keys referencing the objects to be reused as arguments in consecutive function calls, making the individual requests technically stateless. Besides reduced complexity, this system makes parallel computing and asynchronous requests a natural part of the interaction. To compute $f(g(x), h(y))$, the client can perform RPC requests for $g(x)$ and $h(y)$ simultaneously and pass the resulting keys to $f()$ in a second step. In an asynchronous client language such as JavaScript this happens so naturally that it requires almost no effort from the user or application developer.

One important detail is that OpenCPU deliberately does not prescribe how the server should implement storing and loading of objects in between requests. The API only specifies a system for performing R function calls over HTTP and referencing objects from keys. Different server implementations can use different strategies for retaining such objects. A naive implementation could simply serialize objects to disk after each request and immediately terminate the process. This is safe and easy, but writing to disk can be slow. A more sophisticated implementation could keep objects in memory for a while longer, either by keeping the R process alive or through some sort of in-memory database or memcached system. Thereby the resources do not need to be loaded from disk if they are reused in a subsequent request shortly after being created. This illustrates the kind of optimization that can be achieved by carefully decoupling server and client components.

2.4 The OpenCPU HTTP API

This section introduces the most important concepts and operations of the API. At this point the concerns discussed in earlier chapters become more concrete as we illustrate how the pieces come together in the context of R and HTTP. It is not the intention to provide a detailed specification of every feature of the system. We focus on the main parts of the interface that exemplify the separation of concerns central to this work. The online documentation and reference implementations are the best source of information on the specifics of implementing clients and applications.

2.4.1 About HTTP

One of the major strengths of `OpenCPU` is that it builds on the hypertext transfer protocol (Fielding et al., 1999). `HTTP` is the most used application protocol on the internet, and the foundation of data communication in browsers and the world wide web. The `HTTP` specification is very mature and widely implemented. It provides all functionality required to build modern applications and has recently gained popularity for web `API`'s as well. The benefit of using a standardized application protocol is that a lot of functionality gets built-in by design. `HTTP` has excellent mechanisms for authentication, encryption, caching, distribution, concurrency, error handling, etc. This allows us to defer most application logic of our system to the protocol and limit the `API` specification to logic of scientific computing.

The `OpenCPU API` defines a mapping between `HTTP` requests and high-level operations such as calling functions, running scripts, access to data, manual pages and management of files and objects. The `API` deliberately does not prescribe any language implementation details. Syntax and low-level concerns such as process management or code evaluation are abstracted and at the discretion of the server implementation. The `API` also does not describe any logic which can be taken care of on the protocol or application layer. For example, to add support for authentication, any of the standard mechanisms can be used such as `basic auth` (Franks et al., 1999) or `OAuth 2.0` (Hardt, 2012). The implementation of such authentication methods might vary from a simple server configuration to defining additional endpoints. But because authentication will not affect the meaning of the `API` itself, it can be considered independent of this research. The same holds for other features of the HTTP protocol which can be used in conjunction with the OpenCPU API (or any other HTTP interface for that matter). What remains after cutting out implementation and application logic is a simple and interoperable interface that is easy to understand and can be implemented with standard HTTP software libraries. This is an enormous advantage over many other bridges to R and critical to make the system scalable and extensible.

2.4.2 Resource types

As was described earlier, individual requests within the OpenCPU API are stateless and there is no notion of a *process*. State of the system changes through creation and manipulation of resources. This makes the various resource types the conceptual building blocks of the API. Each resource type has unique properties and supports different operations.

2.4.2.1 Objects

Objects are the main entities of the system and carry the same meaning as within a functional language. They include data structures, functions, or other types supported by the back-end language, in this case R. Each object has an individual endpoint within the API and unique name or key within its namespace. The client needs no knowledge of the implementation of these objects. Analogous to a UI, the primary purpose of the API is managing objects (creating, retrieving, publishing) and performing procedure calls. Objects created from executing a script or returned by a function call are automatically stored and gain the same status as other existing objects. The API does not distinguish between static objects that appear in e.g. packages, or dynamic objects created by users, nor does it distinguish between objects in memory or on disk. The API merely provides a system for referencing objects in a way that allows clients to control and reuse them. The implementation of persistence, caching and expiration of objects is at the discretion of the server.

2.4.2.2 Namespaces

A namespace is a collection of uniquely named objects with a given path in the `API`. In `R`, static namespaces are implemented using *packages* and dynamic namespaces exist in *environments* such as the user workspace. `OpenCPU` abstracts the concept of a namespace as a set of uniquely named objects and does not distinguish between static, dynamic, persistent or temporary namespaces. Clients can request a list of the contents of any namespace, yet the server might refuse such a request for private namespaces or hidden objects.

2.4.2.3 Formats

`OpenCPU` explicitly differentiates a resource from a *representation* of that resource in a particular *format*. The `API` lets the client rather than the server decide on the format used to serve content. This is a difference with common scientific practices of exchanging data, documents and figures in fixed format files. Resources in `OpenCPU` can be retrieved using various output formats and formatting parameters. For example, a basic dataset can be retrieved in `csv`, `json`, `Protocol Buffers` or `tab delimited` format. Similarly, a graphic can be retrieved in `svg`, `png` or `pdf` and manual pages can be retrieved in `text`, `html` or `pdf` format. In addition to the format, the client can specify formatting parameters in the request. The system supports many additional formats, but not every format is appropriate for every resource type. When a client requests a resource in a format using an invalid format, the server responds with an error.

2.4.2.4 Data

The `API` defines a separate entity for *data* objects. Even though data can technically be treated as general objects, they often serve a different purpose. Data are usually not language specific and cannot be called or executed. Therefore it can be useful to conceptually distinguish this subclass. For example, `R` uses lazy loading of data objects to save memory when for packages containing large datasets.

2.4.2.5 Graphics

Any function call can produce zero or more graphics. After completing a remote function call, the server reports how many graphics were created and provides the key for referencing these graphics. Clients can retrieve each individual graphic in subsequent requests using one of various output formats such as `png`, `pdf`, and `svg`. Where appropriate the client can specify additional formatting parameters during the retrieval of the graphic such as width, height or font size.

2.4.2.6 Files

Files can be uploaded and downloaded using standard `HTTP` mechanics. The client can post a file as an argument in a remote function call, or download files that were saved to the working directory by the function call. Support for files also allows for hosting web pages (e.g. `html`, `css`, `js`) that interact with local `API` endpoints to serve a web application. Furthermore files that are recognized as *scripts* can be executed using `RPC`.

2.4.2.7 Manuals

In most scientific computing languages, each function or dataset that is available to the user is accompanied by an identically named manual page. This manual page includes information such as description and usage of functions and their arguments, or comments about the columns of a particular dataset. Manual pages can be retrieved through the `API` in various formats including `text`, `html` and `pdf`.

2.4.2.8 Sources

The `OpenCPU` specification makes reproducibility an integrated part of the `API` interaction. In addition to results, the server stores the call and arguments for each `RPC` request. The same key that is used to retrieve objects or graphics can be used to retrieve sources or automatically replicate the computation. Hence for each output resource on the system, clients can lookup the code, data, warnings and packages that were involved in its creation. Thereby results can easily be recalculated, which forms a powerful basis for reproducible practices. This feature can be used for other purposes as well. For example, if a function fetches dynamic data from an external resource to generate a model or plot, reproduction is used to *update* the model or plot with new data.

2.4.2.9 Containers

We refer to a path on the server containing one or more collections of resources as a *container*. The current version of `OpenCPU` implements two types of containers. A *package* is a static container which may include a namespace with `R` objects, manual pages, data and files. A *session* is a dynamic container which holds outputs created from executing a script or function call, including a namespace with `R` objects, graphics and files. The distinction between packages and sessions is an implementation detail. The `API` does not differentiate between the various container types: interacting with an object or file works the same, regardless of whether it is part of a package or session. Future versions or other servers might implement different container types for grouping collections of resources.

2.4.2.10 Libraries

We refer to a collection of containers as a *library*. In `R` terminology, a library is a directory on disk with installed packages. Within the context of the `API`, the concept is not limited to packages but refers more generally to any set of containers. The `/ocpu/tmp/` library for example is the collection of temporary sessions. Also the `API` notion of a library does not require containers to be preinstalled. A remote collection of packages, which in `R` terminology is called a *repository*, can also be implemented as a library. The current implementation of `OpenCPU` exposes the `/ocpu/cran/` library which refers to the current packages on the `CRAN` repository. The `API` does not differentiate between a library of sessions, local packages or remote packages. Interacting with an object from a `CRAN` package works the same as interacting with an object from a local package or temporary session. The `API` leaves it up to the server which types of libraries it wishes to expose and how to implement this. The current version of `OpenCPU` uses a combination of cron-jobs and on-the-fly package installations to synchronize packages on the server with the `CRAN` repositories.

2.4.3 Methods

The current `API` uses two `HTTP` methods: `GET` and `POST`. As per `HTTP` standards, `GET` is a *safe* method which means it is intended only for information reading and should not change the state of the server. `OpenCPU` uses the `GET` method to retrieve objects, manuals, graphics or files. The parameters of the request are mapped to the formatting function. A `GET` requests targeting a container, namespace or directory is used to list the contents. The `POST` method on the other is used for `RPC` which does change server state. A `POST` request targeting a function results in a remote function call where the `HTTP` parameters are mapped to function arguments. A `POST` request targeting a script results in an execution of the script where HTTP parameters are mapped to the script interpreter. Table 2.1 gives an overview using the MASS package (Venables and Ripley, 2002) as an example.

2.4.4 Status codes

Each HTTP response includes a status code. Table 2.2 lists some common HTTP status codes used by OpenCPU that the client should be able to interpret. The meaning of these status codes is conform HTTP standards. The web server may use additional status codes for more general purposes that are not specific to OpenCPU.

2.4.5 Content-types

Clients can retrieve objects in various *formats* by adding a format identifier suffix to the URL in a GET request. Which formats are supported and how object types map to a particular format is at the discretion of the server implementation. Not every format can support any object type. For example, `csv` can only be used to retrieve tabular data structures and `png` is only appropriate for graphics. Table 2.3 lists the formats OpenCPU supports, the respective internet media type, and the R function that OpenCPU uses to export an object into a particular format. Arguments of the GET requests are mapped to this export function. The `png` format has parameters such as `width` and `height` as documented in `?png`, whereas the `tab` format has parameters `sep`, `eol`, `dec` which specify the delimiting, end-of-line and decimal character respectively as documented in `?write.table`.

2.4.6 URLs

The root of the API is dynamic, but defaults to `/ocpu/` in the current implementation. Clients should make the OpenCPU server address and root path configurable.

In the examples we assume the defaults. As discussed before, `OpenCPU` currently implements two container types to hold resources. Table 2.4 lists the `URL`s of the *package* container type, which includes objects, data, manual pages and files.

Table 2.5 lists `URL`s of the *session* container type. This container holds outputs generated from a `RPC` request and includes objects, graphics, source code, stdout and files. As noted earlier, the distinction between packages and sessions is considered an implementation detail. The `API` does not differentiate between objects and files that appear in packages or in sessions.

2.4.7 RPC requests

A `POST` request in `OpenCPU` always invokes a remote procedure call (`RPC`). Requests targeting a *function* object result in a function call where the `HTTP` parameters from the post body are mapped to function *arguments*. A `POST` targeting a *script* results in execution of the script where `HTTP` parameters are passed to the script interpreter. The term `RPC` refers to both remote function calls and remote script executions. The current `OpenCPU` implementation recognizes scripts by their file extension, and supports `R`, `latex`, `markdown`, `Sweave` and `knitr` scripts. Table 2.6 lists each script type with the respective file extension and interpreter.

An important conceptual difference with a terminal interface is that in the `OpenCPU API`, the server determines the namespace that output of a function call is assigned to. The server includes a *temporary key* in the `RPC` response that serves the same role as a variable name. The key and is used to reference the newly created resources in future requests. Besides the return value, the server also stores graphics, files, warnings, messages and `stdout` that were created by the `RPC`. These can be listed and retrieved using the same key. In `R`, the function call itself is also an object which is added to the collection for reproducibility purposes.

Objects on the system are non-mutable and therefore the client cannot change or overwrite existing keys. For functions that modify the state of an object, the server creates a copy of the modified resource with a new key and leaves the original unaffected.

2.4.8 Arguments

Arguments to a remote function call can be posted using one of several methods. A data interchange format such as `JSON` or `Protocol Buffers` can be used to directly post data structures such as lists, vectors, matrices or data frames. Alternatively the client can reference the name or key of an existing object. The server automatically resolves keys and converts interchange formats into objects to be used as arguments in the function call. Files contained in a `multipart/form-data` payload of an `RPC` request are copied to the working directory and the argument of the function call is set to the filename. Thereby, remote function calls with a file arguments can be performed using standard `HTML` form submission.

The current implementation supports several standard `Content-type` formats for passing arguments to a remote function call within a `POST` request, including `application/x-www-form-urlencoded`, `multipart/form-data`, `application/json` and `application/x-protobuf`. Each parameter or top level field within a `POST` payload contains a single argument value. Table 2.7 shows a matrix supported argument formats for each `Content-types`.

2.4.9 Privacy

Because the data and sources of a statistical analysis include potentially sensitive information, the temporary keys from `RPC` requests are private. Clients should default to keeping these keys secret, given that leaking a key will compromise confidentiality of their data. The system does not allow clients to search for keys or retrieve resources without providing the appropriate key. In this sense, a temporary key has a similar status as an *access token*. Because temporary keys are private, multiple users can share a single `OpenCPU` server without any form of authentication. Each request is anonymous and confidential, and only the client that performed the `RPC` has the key to access resources from a particular request.

However, temporary keys do not have to be kept private per se: clients can choose to exchange keys with other clients. Unlike typical access tokens, the keys in `OpenCPU` are unique for each request. Hence by publishing a particular key, the client reveals only the resources from a specific `RPC` request, and no other confidential information. Resources in `OpenCPU` are not tied to any particular user, in fact, there are no users in `OpenCPU` system itself. Clients can share objects, graphics or files with each other, simply by communicating keys to these resources. Because each key holds both the output as well as the sources for an `RPC` request, shared objects are reusable and reproducible by design. In some sense, all clients share a single universal namespace with keys containing hidden objects from all `RPC` requests. By knowing a key to a particular resource it can be used as any other object on the system. This shapes the contours of a social analysis platform in which users collaborate by sharing reproducible, reusable resources identified by unique keys.

Method	*Target*	*Action*	*Parameters*	*Example*
`GET`	object	retrieve	formatting	`GET /ocpu/library/MASS/data/cats/json`
	manual	read	formatting	`GET /ocpu/library/MASS/man/rlm/html`
	graphic	render	formatting	`GET /ocpu/tmp/{key}/graphics/1/png`
	file	download	-	`GET /ocpu/library/MASS/NEWS`
	path	list contents	-	`GET /ocpu/library/MASS/scripts/`
`POST`	object	call function	function arguments	`POST /ocpu/library/stats/R/rnorm`
	file	run script	control interpreter	`POST /ocpu/library/MASS/scripts/ch01.R`

Table 2.1: Currently implemented `HTTP` methods

Status Code	*Happens when*	*Response content*
`200 OK`	On successful `GET` request	Requested data
`201 Created`	On successful `POST` request	Output key and location
`302 Found`	Redirect	Redirect location
`400 Bad Request`	On computational error in `R`	Error message from `R` in `text/plain`
`502 Bad Gateway`	Back-end server offline	– (See error logs)
`503 Bad Request`	Back-end server failure	– (See error logs)

Table 2.2: Commonly used `HTTP` status codes

Format	*Content-type*	*Export function*	*Example*
`print`	`text/plain`	`base::print`	`/ocpu/cran/MASS/R/rlm/print`
`rda`	`application/octet-stream`	`base::save`	`/ocpu/cran/MASS/data/cats/rda`
`rds`	`application/octet-stream`	`base::saveRDS`	`/ocpu/cran/MASS/data/cats/rds`
`json`	`application/json`	`jsonlite::toJSON`	`/ocpu/cran/MASS/data/cats/json`
`pb`	`application/x-protobuf`	`RProtoBuf::serialize_pb`	`/ocpu/cran/MASS/data/cats/pb`
`tab`	`text/plain`	`utils::write.table`	`/ocpu/cran/MASS/data/cats/tab`
`csv`	`text/csv`	`utils::write.csv`	`/ocpu/cran/MASS/data/cats/csv`
`png`	`image/png`	`grDevices::png`	`/ocpu/tmp/{key}/graphics/1/png`
`pdf`	`application/pdf`	`grDevices::pdf`	`/ocpu/tmp/{key}/graphics/1/pdf`
`svg`	`image/svg+xml`	`grDevices::svg`	`/ocpu/tmp/{key}/graphics/1/svg`

Table 2.3: Currently supported export formats and corresponding `Content-type`

Path	*Description*	*Examples*
.	Package information	`/ocpu/cran/MASS/`
`./R`	Exported namespace objects	`/ocpu/cran/MASS/R/`
		`/ocpu/cran/MASS/R/rlm/print`
`./data`	Data objects in the package (`HTTP GET` only)	`/ocpu/cran/MASS/data/`
		`/ocpu/cran/MASS/data/cats/json`
`./man`	Manual pages in the package (`HTTP GET` only)	`/ocpu/cran/MASS/man/`
		`/ocpu/cran/MASS/man/rlm/html`
`./*`	Files in installation directory, relative to package the root	`/ocpu/cran/MASS/NEWS`
		`/ocpu/cran/MASS/scripts/`

Table 2.4: The package container includes objects, data, manual pages and files.

Path	*Description*	*Examples*
`.`	Session content list	`/ocpu/tmp/{key}/`
`./R`	Objects created by the `RPC` request	`/ocpu/tmp/{key}/R/`
		`/ocpu/tmp/{key}/R/mydata/json`
`./graphics`	Graphics created by the `RPC` request	`/ocpu/tmp/{key}/graphics/`
		`/ocpu/tmp/{key}/graphics/1/png`
`./source`	Source code of `RPC` request	`/ocpu/tmp/{key}/source`
`./stdout`	`STDOUT` from by the `RPC` request	`/ocpu/tmp/{key}/stdout`
`./console`	Mixed source and `STDOUT` emulating console output	`/ocpu/tmp/{key}/console`
`./files/*`	Files saved to working dir by the `RPC` request	`/ocpu/tmp/{key}/files/myfile.xyz`

Table 2.5: The session container includes objects, graphics, source, stdout and files.

File extension	*Type*	*Interpreter*
`file.r`	`R`	`evaluate::evaluate`
`file.tex`	LaTeX	`tools::texi2pdf`
`file.rnw`	`knitr`/`sweave`	`knitr::knit` + `tools::texi2pdf`
`file.md`	`markdown`	`knitr::pandoc`
`file.rmd`	`knitr markdown`	`knitr::knit` + `knitr::pandoc`
`file.brew`	`brew`	`brew::brew`

Table 2.6: Files recognized as scripts and their characterizing file extension

Content-type	*Primitives*	*Data structures*	*Raw code*	*Files*	*Temp key*
`multipart/form-data`	OK	OK (inline `json`)	OK	OK	OK
`application/x-www-form-urlencoded`	OK	OK (inline `json`)	OK	-	OK
`application/json`	OK	OK	-	-	-
`application/x-protobuf`	OK	OK	-	-	-

Table 2.7: Accepted request `Content-types` and supported argument formats

CHAPTER 3

The `RAppArmor` Package: Enforcing Security Policies in `R` Using Dynamic Sandboxing on Linux

3.1 Security in `R`: introduction and motivation

The `R` project for statistical computing and graphics (R Development Core Team, 2012) is currently one of the primary tool-kits for scientific computing. The software is widely used for research and data analysis in both academia and industry, and is the de-facto standard among statisticians for the development of new computational methods. With support for all major operating systems, a powerful standard library, over 3000 add-on packages and a large active community, it is fair to say that the project has matured to a production-ready computation tool. However, practices in statistical computation have changed since the initial design of `R` in 1993 (Ihaka, 1998). Internet access, public cloud computing (Armbrust et al., 2010), live and open data and scientific super computers are transforming the landscape of data analysis. This is only the beginning. Sharing of data, code and results on social computing platforms will likely become an integral part of the publication process (Stefanski et al., 2013). This could address some of the hardware challenges, but also contribute towards reproducible research and further socialize data analysis, i.e. facilitate learning, collaboration and integration. These developments shift the role of statistical software towards more general purpose computational back-ends, powering systems and applications with embedded analytics and visualization.

However, one reason developers might still be reluctant to build on `R` is concerns regarding security and management of shared hardware resources. Reliable software systems require components which behave predictably and cannot be abused. Because `R` was primarily designed with the local user in mind, security restrictions and unpredictable behavior have not been considered a major concern in the design of the software. Hence, these problems will need to be addressed somehow before developers can feel comfortable making `R` part of their infrastructure, or convince administrators to expose their facilities to the public for `R` based services. It is our personal experience that the complexity of managing security is easily underestimated when designing stacks or systems that build on `R`. Some of the problems are very domain specific to scientific computing, and make embedding `R` quite different from embedding other software environments. Properly addressing these challenges can help facilitate wider adoption of `R` as a general purpose statistical engine.

3.1.1 Security when using contributed code

Building systems on `R` has been the main motivation for this research. However, security is a concern for `R` in other contexts as well. As the community is growing rapidly, relying on social courtesy in contributed code becomes more dangerous. For example, on a daily basis, dozens of packages and package updates submitted to the *Comprehensive R Archive Network* (CRAN) (Ripley, 2011). These packages contain code written in `R`, `C`, `Fortran`, `C++`, `Java`, etc. It is unfeasible for the CRAN maintainers to do a thorough audit of the full code that is submitted, every time. Some packages even contain pre-compiled `Java` code for which the source is not included. Furthermore, `R` packages are not signed with a private key as is the case for e.g. packages in most `Linux` distributions, which makes it hard to verify the identity of the author. As CRAN packages are automatically build and installed on hundreds, possibly thousands of machines around the world, they form an interesting target for abuse. Hence there is a real dangler of packages containing malicious code making their way unnoticed into the repositories. Risks are even greater for packages distributed through channels without any form of code review, for example by email or through the increasingly popular Github repositories (Torvalds and Hamano, 2010; Dabbish et al., 2012).

In summary, it is not overly paranoid of the `R` user to be a bit cautious when installing and running contributed code downloaded from the internet. However, things don't have to be as critical as described above. Controlling security is a good practice, even when there are no immediate reasons for concern. Some users simply might want to prevent `R` from accidentally erasing files or interfering with other activities on the machine. Making an effort to ensure `R` is running safely with no unnecessary privileges can be reassuring to both user and system administrator, and might one day prevent a lot of trouble.

3.1.2 Sandboxing the `R` environment

This paper explores some of the potential problems, along with approaches and methods of securing `R`. Different aspects of security in the context of `R` are illustrated using personal experiences and examples of bad or malicious code. We will explain how untrusted code can be executed inside a *sandboxed* process. Sandboxing in this context is a somewhat informal term for creating an execution environment which limits capabilities of harmful and undesired behavior. As it turns out, `R` itself is not very suitable for implementing such access control policies, and the only way to properly enforce security is by leveraging features from the operating system. To exemplify this approach, an implementation based on `AppArmor` is provided which can be used on `Linux` distributions as the basis for a sandboxing toolkit. This package is used throughout the paper to demonstrate one way of addressing issues.

However, we want to emphasize that we don't claim to have solved the problem. This paper mostly serves an introduction to security for the R community, and hopefully creates some awareness that this is a real issue moving forward. The RAppArmor package is one approach and a good starting point for experimenting with dynamic sandboxing in R. However it mostly serves as a proof of concept of the general idea of confining and controlling an R process. The paper describes examples of use cases, threats, policies and anecdotes to give the reader a sense of what is involved with this topic. Without any doubt, there are concerns beyond the ones mentioned in this paper, many of which might be specific to certain applications or systems. We hope to invoke a discussion in the community about potential security issues related to using R in different scenarios, and encourage those comfortable with other platforms or who use R for different purposes to join the discussion and share their concerns, experiences and solutions.

3.2 Use cases and concerns of sandboxing R

Let us start by taking a step back and put this research in perspective by describing some concrete use cases where security in R could be a concern. Below three simple examples of situations in which running R code in a sandbox can be useful. The use cases are ordered by complexity and require increasingly advanced sandboxing technology.

3.2.0.1 Running untrusted code

Suppose we found an R package in our email or on the internet that looks interesting, but we are not quite sure who the author is, and if the package does not contain any malicious code. The package is too large for us to inspect all of the code manually, and furthermore it contains a library in a foreign language (e.g. C++, Fortran) for which we lack the knowledge and insight to really understand its implications. Moreover, programming style (or lack thereof) of the author can make it difficult to assess what exactly is going on (IOCCC, 2012). Nevertheless we would like to give the package a try, but without exposing ourselves to the risk of potentially jeopardizing the machine.

One solution would be to run untrusted code on a separate or virtual machine. We could either install some local virtualization software, or rent a VPS/cloud server to run R remotely, for example on Amazon EC2. However this is somewhat cumbersome and we will not have our regular workflow available: in order to put the package to the test on our own data, we first need to copy our data, scripts, files and package library, etc. Some local virtualization software can be configured for read-only sharing of resources between host and guest machine, but we would still need separate virtual machines for different tasks and projects. In practice, managing multiple machines is a bit unpractical and not something that we might want to do on a daily basis. It would be more convenient if we could instead sandbox our regular `R` environment for the duration of installing and using the new package with a tailored security policy. If the sandbox is flexible and unobtrusive enough not to interfere with our daily workflow, we could even make a habit out of using it each time we use contributed code (which to most users means every day).

3.2.0.2 Shared resources

A second use case could be a scenario where multiple users are sharing a single machine. For example, a system administrator at a university is managing a large computing server and would like to make it available to faculty and students. This would allow them to run `R` code that requires more computing power than their local machine can handle. For example a researcher might want to do a simulation study, and fit a complex model a million times on generated datasets of varying properties. On her own machine this would take months to complete, but the super computer can finish the job overnight. The administrator would like to set up a web service for this and other researchers to run such R scripts. However he is worried about users interfering with each others work, or breaking anything on the machine. Furthermore he wants to make sure that system resources are allocated in a fair way such that no single user can consume all memory or cpu on the system.

3.2.0.3 Embedded systems and services

There have numerous efforts to facilitate integration of `R` functionality into 3rd party systems, both for open source and proprietary purposes. Major commercial vendors like Oracle, IBM and SAS have included `R` interfaces in their products. Examples of open source interfaces from popular general purpose languages are `RInside` (Eddelbuettel and Francois, 2011a), which embeds `R` into `C++` environments, and `JRI` which embeds `R` in `Java` software (Urbanek, 2011, 2013c). Similarly, `rpy2` (Moreira and Warnes, 2006; Gautier, 2012) provides a `Python` interface to `R`, and `RinRuby` is a `Ruby` library that integrates the `R` interpreter in Ruby (Dahl and Crawford, 2009). Littler provides hash-bang (i.e. script starting with `#!/some/path`) capability for `R` (Horner and Eddelbuettel, 2011). The Apache2 module rApache (`mod_R`) (Horner, 2011) makes it possible to run `R` scripts from within the Apache2 web server. Heiberger and Neuwirth (2009) provide a series of tools to call `R` from DCOM clients on Windows environments, mostly to support calling `R` from Microsoft Excel. Finally, `RServe` is TCP/IP server which provides low level access to an `R` session over a socket (Urbanek, 2013a).

The third use case originates from these developments: it can be summarized as confining and managing `R` processes inside of embedded systems and services. This use case is largely derived from our personal needs: we are using `R` inside various systems and web services to provide on-demand calculating and plotting over the internet. These services need to respond quickly and with minimal overhead to incoming requests, and should scale to serve many jobs per second. Furthermore the systems need to be stable, requiring that jobs should always return within a given timeframe. Depending on user and the type of job, different security restrictions might be appropriate. Some services specifically allow for execution of arbitrary `R` code. Also we need to dynamically enforce limits on the use of memory, processors and disk space on a per process basis. These requirements demand a more flexible and finer degree of control over the process privileges and restrictions than the first two use cases. It encouraged us to explore more advanced methods than the conventional tools and has been the most central motivation of this research.

3.2.1 System privileges and hardware resources

The use cases described above outline motivations and requirements for an `R` sandbox. Two inter-related problems can be distinguished. The first one is preventing system abuse, i.e. use of the machine for malicious or undesired activity, or completely compromising the machine. The second problem is managing hardware resources, i.e. preventing excessive use by constraining the amount of memory, cpu, etc that a single user or process is allowed to consume.

3.2.1.1 System abuse

The `R` console gives the user direct access to the operating system and does not implement any privilege restrictions or access control policies to prevent malicious use. In fact, some of the basic functionality in `R` assumes quite profound access to the system, e.g. read access to system files, or the privilege of running system shell commands. However, running untrusted `R` code without any restrictions can get us in serious trouble. For example, the code could call the `system()` function from where any shell commands can be executed. But also innocent looking functions like `read.table` can be used to extract sensitive information from the system, e.g. `read.table("/etc/passwd")` lists all users on the system or `readLines("/var/log/syslog")` exposes system log information.

Even an `R` process running as a non-privileged user can do a lot of harm. Potential perils include code containing or downloading a virus or security exploit, or searching the system for sensitive personal information. Appendix B.2 demonstrates a hypothetical example of a simple function that scans the home directory for documents containing credit card numbers. Another increasing global problem are viruses which make the machine part of a so called "botnet". Botnets are large networks of compromised machines ("bots") which are remotely controlled to used for illegal activities (Abu Rajab et al., 2006). Once infected, the botnet virus connects to a centralized server and waits for instructions from the owner of the botnet. Botnets are mostly used to send spam or to participate in DDOS attacks: centrally coordinated operations in which a large number of machines on the internet is used to flood a target with network traffic with the goal of taking it down by overloading it (Mirkovic and Reiher, 2004). Botnet software is often invisible to the user of an infected machine and can run with very little privileges: simple network access is sufficient to do most of its work.

When using `R` on the local machine and only running our own code, or from trusted sources, these scenarios might sound a bit far fetched. However, when running code downloaded from the internet or exposing systems to the public, this is a real concern. Internet security is a global problem, and there are a large number of individuals, organizations and even governments actively employing increasingly advanced and creative ways of gaining access to protected infrastructures. Especially servers running on beefy hardware or fast connections are attractive targets for individuals that could use these resources for other purposes. But also servers and users inside large companies, universities or government institutions are frequently targeted with the goal of gathering confidential information. This last aspect seems especially relevant, as `R` is used frequently in these types of organizations.

3.2.1.2 Resource restrictions

The other category of problems is not necessarily related to deliberate abuse, and might even arise completely unintentionally. It involves proper management, allocation and restricting of hardware.

It is fair to say that `R` can be quite greedy with system resources. It is easy to run a command which will consume all of the available memory and/or CPU, and does not finish executing until manually terminated. When running `R` on the local machine through the interactive console, the user will quickly recognize a function call that is not returning timely or is making the machine unresponsive. When this happens, we can easily interrupt the process prematurely by sending a `SIGINT`, i.e. pressing `CTRL+C` in `Linux` or `ESC` in Windows. If this doesn't work we can open the task manager and tell the operating system to kill the process, and if worst comes to worst we can decide to reboot our machine.

However, when `R` is embedded in a system, the situation is more complicated and we need to cover these scenarios in advance. If an out-of-control `R` job is not properly detected and terminated, the process might very well run indefinitely and take down our service, or even the entire machine. This has actually been a major problem that we personally experienced in an early implementation of a public web service for mixed modelling (Ooms, 2010) which uses the `lme4` package (Bates et al., 2011). What happened was that users could accidentally specify a variable with many levels as the *grouping factor*. This causes the design matrix to blow up even on a relatively small dataset, and decompositions will take forever to complete. To make things worse, `lme4` uses a lot of `C` code which does not respond to time limits set by R's `setTimeLimit` function. Appendix B.4 contains a code snippet that simulates this scenario. When this would happen, the only way to get things up and running again was to manually login to the server and reset the application.

Unfortunately this example is not an exception. The behavior of `R` can be unpredictable, which is an aspect easily overlooked by (non-statistician) developers. When a system calls out to e.g. an `SQL` or `PHP` script, the procedure usually runs without any problems and the processing time is proportional to the size of the data, i.e. the number of records returned by `SQL`. However, in an `R` script many things can go wrong, even though the script itself is perfectly fine. Algorithms might not converge, data might be rank-deficient, or missing values throw a spanner in the works. Statistical computing is full of such intrinsic edge-cases and unexpected caveats. Using only tested code or predefined services does not entirely guarantee smooth and timely completion of `R` jobs, especially if the data is dynamic. When embedding `R` in systems or shared facilities, it is important that we acknowledge this facet and have systems in place to manage jobs and mitigate problems without manual intervention.

3.3 Various approaches of confining `R`

The current section introduces several approaches of securing and sandboxing `R`, with their advantages and limitations. They are reviewed in the context of our use cases, and evaluated on how they address the problems of system abuse and restricting resources. The approaches are increasingly *low-level*: they represent security on the level of the application, R software itself and operating system respectively. As will become clear, we are leaning towards the opinion that `R` is not very well suited to address security issues, and the only way to do proper sandboxing is on the level of the operating system. This will lead up to the `RAppArmor` package introduced in section 3.4.

3.3.1 Application level security: predefined services

The most common approach to prevent malicious use is simply to only allow a limited set of predefined services, that have been deployed by a trusted developer and cannot be abused. This is generally the case for websites containing dynamic content though e.g. `CGI` or `PHP` scripts. Running arbitrary code is explicitly prevented and any possibility to do so anyway is considered a security hole. For example, we might expose the following function as a web service:

```r
liveplot <- function (ticker) {
  url <- paste("http://ichart.finance.yahoo.com/table.csv?s=",
    ticker, "&a=07&b=19&c=2004&d=07&e=13&f=2020&g=d&ignore=.csv",
    sep = "")
  mydata <- read.csv(url)
  mydata$Date <- as.Date(mydata$Date)
  myplot <- ggplot2::qplot(Date, Close, data = mydata, geom = c("line",
    "smooth"), main = ticker)
  print(myplot)
}
```

The function above downloads live data from the public API at Yahoo Finance and creates an on-demand plot of the historical prices using `ggplot2` (Wickham, 2009). It has only one parameter: `ticker`, a character string identifying a stock symbol. This function can be exposed as a predefined web service, where the client only supplies the `ticker` argument. Hence the system does not need to run any potentially harmful user-supplied `R` code. The client sets the symbol to e.g. `"GOOG"` and the resulting plot can be returned in the form of a PNG image or PDF document. This function is actually the basis of the "stockplot" web application (Ooms, 2009); an interactive graphical web application for financial analysis which still runs today.

Limiting users and clients to a set of predefined parameterized services is the standard solution and reasonably safe in combination with basic security methods. For example, `Rserve` can be configured to run with a custom `uid`, `umask`, `chroot`. However in the context of `R`, predefined services severely limit the application and security is actually not fully guaranteed. We can expose some canned calculations or generate a plot as done in the example, but beyond that things quickly becomes overly restrictive. For example in case of an application that allows the user to fit a statistical model, the user might need to be able to include transformations of variables like `I(cos(x^ 2))` or `cs(x, 3)`. Not allowing a user to call any custom functions makes this hard to implement.

What distinguishes `R` from other statistical software is that the user has a great deal of control and can do programming, load custom libraries, etc. A predefined service takes this freedom away from the user, and at the same time puts a lot of work in the hands of the developer and administrator. Only they can expose new services and they have to make sure that all services that are exposed cannot be abused in some way or another. Therefore this approach is expensive, and not very social in terms of users contributing code. In practice, anyone that wants to publish an `R` service will have to purchase and manage a personal server or know someone that is willing to do so. Also it is still important to set hardware limitations, even when exposing relatively simple, restricted services. We already mentioned the example of the `lme4` web application, where a single user could accidentally take down the entire system by specifying an overly complex model. But actually even some of the most basic functionality in `R` can cause trouble with problematic data. Hence, even restricted predefined `R` services are not guaranteed to consistently return smooth and timely. These aspects of statistical computing make common practices in software design not directly generalize to `R` services, and are easily under appreciated by developers and engineers with a limited background in data analysis.

3.3.1.1 Code injection

Finally, there is still the risk of *code injection*. Because `R` is a very dynamic language, evaluations sometimes happen at unexpected places. One example is during the parsing of `formulas`. For example, we might want to publish a service that calls the `lm()` function in `R` on a fixed dataset. Hence the only parameter supplied by the user is a *formula* in the form of a character vector. Assume in the code snippet below that the `userformula` parameter is a string that has been set through some graphical interface.

```r
coef(lm(userformula, data = cars))
```

For example the user might supply a string `"speed~dist"` and the service will return the coefficients. On first sight, this might seem like a safe service. However, formulas actually allow for the inclusion of calls to other functions. So even though `userformula` is a character vector, it can be used to inject a function call:

```r
userformula <- "speed ~ dist + system('whoami')"
lm(userformula, data = cars)
```

In the example above, `lm` will automatically convert `userformula` from type character to a `formula`, and subsequently execute the `system('whoami')` command. Hence even when a client supplies only simple primitive data, unexpected opportunities for code injection can still arise. Therefore it is important when using this approach, to sanitize the input before executing the service. One way is by setting up the service such that only alphanumeric values are valid parameters, and use a regular expression to remove any other characters, before actually executing the script or service:

```r
myarg <- gsub("[^a-zA-Z0-9]", "", myarg)
```

3.3.2 Sanitizing code by blacklisting

A less restrictive approach is to allow users to push custom R code, but inspect the code before evaluating it. This approach has been adopted by some web sites that allow users to run R code, like Banfield (1999) and Chew (2012). However, given the dynamic nature of the R language, malicious calls are actually very difficult to detect and such security is easy to circumvent. For example, we might want to prevent users from calling the `system` function. One way is to define some smart regular expressions that look for the word "system" in a block of code. This way it would be possible to detect a potentially malicious call like this:

```r
system("whoami")
```

However, it is much more difficult to detect the equivalent call in the following block:

```r
foo <- get(paste("sy", "em", sep = "st"))
bar <- paste("who", "i", sep = "am")
foo(bar)
```

And indeed, it turns out that the services that use this approach are fairly easy to trick. Because R is a dynamic scripting language, the exact function calls might not reveal themselves until runtime, when it is often too late. We are actually quite convinced that it is nearly impossible to really sanitize an R script just by inspecting the source code.

An alternative method to prevent malicious code is by defining an extensive blacklist of functions that a user is not allowed to call, and disable these at runtime. The `sandboxR` package (Daroczi, 2013) does this to block access to all R functions providing access to the file system. It evaluates the user-supplied code in an environment in which all blacklisted functions are masked from the calling namespace. This is fairly effective and provides a barrier against smaller possible attacks or casual errors. However, the method relies on exactly knowing and specifying which functions are *safe* and which are not. The package author has done this for the thousands of `R` functions in the base package and we assume he has done a good job. But it is quite hard to maintain and cumbersome to generalize to other `R` packages (by default the method does not allow loading other packages). Everything falls if one function has been overlooked or changes between versions, which does make the method vulnerable. Furthermore, in `R` even the most primitive functions can be exploited to tamper with scoping and namespaces, so it is unwise to rely solely on this for security.

Moreover, even when sanitizing of the code is successful, this method does not limit the use of hardware resources in any way. Hence, additional methods are still required to prevent excessive use of resources in a public environment. Packages like `sandboxR` should probably only be used to supplement system level security as implemented in the `RAppArmor` package. They can be useful to detect problematic calls earlier on and present informative errors naming a specific forbidden function rather than just "permission denied". But blacklisting solutions are not waterproof and should not be considered a full security solution.

3.3.3 Sandboxing on the level of the operating system

One can argue that managing resources and privileges is something that is outside the domain of the `R` software, and is better left to the operating system. The `R` software has been designed for statistical computing and related functionality; the operating system deals with hardware and security related matters. Hence, in order to really sandbox `R` properly without imposing unnecessary limitations on its functionality, we need to sandbox the *process* on the level of the operating system. When restrictions are enforced by the operating system instead of `R` itself, we do not have to worry about all of the pitfalls and implementation details of `R`. The user can interact freely with `R`, but won't be able to do anything for which the system does not grant permission.

Some operating systems offer more advanced capabilities for setting process restrictions than others. The most advanced functionality is found in `UNIX` like systems, of which the most popular ones are either `BSD` based (`FreeBSD, OSX,` etc) or `Linux` based (`Debian, Ubuntu, Fedora, Suse,` etc). Most `UNIX` like systems implement some sort of `ULIMIT` functionality to facilitate restricting availability of hardware resources on a per-process basis. Furthermore, both `BSD` and `Linux` provide various *Mandatory Access Control* (MAC) systems. On `Linux`, these are implemented as Kernel modules. The most popular ones are `AppArmor` (Bauer, 2006), `SELinux` (Smalley et al., 2001) and `Tomoyo Linux` (Harada et al., 2004). MAC gives a much finer degree of control than standard user-based privileges, by applying advanced security policies on a per-process basis. Using a combination of MAC and `ULIMIT` tools we can do a pretty decent job in sandboxing a single `R` process to a point where it can do little harm to the system. Hence we can run arbitrary `R` code without losing any sleep over potentially jeopardizing the machine. Unfortunately, this approach comes at the cost of portability of the software. As different operating systems implement very different methods for managing processes and privileges, the solutions will be to a large extend OS-specific. In our implementation we have tried to hide these system calls by exposing `R` functions to interact with the kernel. Going forward, eventually these functions could behave somewhat OS specific, abstracting away technicalities and providing similar functionality on different systems. But for now we limit ourselves to systems based on the `Linux` kernel.

3.4 The RAppArmor package

The current section describes some security concepts and how an R process can be sandboxed using a combination of ULIMIT and MAC tools. The methods are illustrated using the RAppArmor package: an implementation based on Linux and AppArmor. AppArmor ("Application Armor") is a security module for the Linux kernel. It allows for associating programs and processes with *security profiles* that restrict the capabilities and permissions of that process. There are two ways of using AppArmor. One is to associate a single, static security profile with every R process. This can be done only by the system administrator and does not require our R package (see also section section 3.4.9). However, this is usually overly restrictive. We want more flexibility to set different policies, priorities and restrictions for different users or tasks.

The RAppArmor package exposes R functions that interface directly to Linux system calls related to setting privileges and restrictions of a running process. Besides applying security profiles, RAppArmor also interfaces to the prlimit call in Linux, which sets RLIMIT (resource limit) values on a process (RLIMIT are the Linux implementation of ULIMIT). Linux defines a number of RLIMIT's, which restrict resources like memory use, number of processes, and stack size. More details on RLIMIT follow in section 3.4.7. Using RAppArmor, the sandboxing functionality is accessible directly from within the R session, without the need for external tools or programs. Once RAppArmor is installed, any user can apply security profiles and restrictions to the running process; no special permissions are required. Furthermore, it allowed us to create the eval.secure function: an abstraction which mimics eval, but has additional parameters to evaluate a single call under a given uid, priority, security policy and resource restrictions.

The RAppArmor package brings the low level system security methods all the way up to level of the R language. Using eval.secure, different parts of our code can run with different security restrictions with minimal overhead, something we call *dynamic sandboxing*. This is incredibly powerful in the context of embedded services, and opens the door applications which explicitly allow for arbitrary code execution; something that previously always had to be avoided for security reasons. This enables a new approach to socialize statistical computing and lies at the core of the `OpenCPU` framework (Ooms, 2013), which exposes a public HTTP API to run and share `R` code on a central server.

3.4.1 `AppArmor` profiles

Security policies are defined in *profiles* which form the core of the `AppArmor` software. A profile consists of a set of rules specified using `AppArmor` syntax in an ascii file. The `Linux` kernel translates these rules to a security policy that it will enforce on the appropriate process. A brief introduction to the `AppArmor` syntax is given in section 3.5.1. The appendix of this paper contains some example profiles that ship with the `RAppArmor` package to get the user started. When the package is installed through the Debian/Ubuntu package manager (e.g. using `apt-get`) the profiles are automatically copied to `/etc/apparmor.d/rapparmor.d/`. Because profiles define file access permissions based the location of files and directories on the file system, they are to some extent specific to a certain `Linux` distribution, as different distributions have somewhat varying conventions on where files are located. The example profiles included with `RAppArmor` are based on the file layout of the `r-base` package (and its dependencies) by Bates and Eddelbuettel (2004) for Debian/Ubuntu, currently maintained by Dirk Eddelbuettel.

The `RAppArmor` package and the included profiles work "out of the box" on Ubuntu 12.04 (Precise) and up, Debian 7.0 (Wheezy) and up. The package can also be used on OpenSuse 12.1 and up, however Suse systems organize the file system in a slightly different way than Ubuntu and Debian, so the profiles need to be modified accordingly. The `RAppArmor` website contains some specific instructions regarding various distributions.

Again, we want to emphasize that the package should mostly be seen as a *reference implementation* to demonstrate how to create a working sandbox in `R`. The `RAppArmor` package provides the tools to set security restrictions and example profiles to get the user started. However, depending on system and application, different policies might be appropriate. It is still up to the administrator to determine which privileges and restrictions are appropriate for a certain system or purpose. The example profiles are merely a starting point and need fine-tuning for specific applications.

3.4.2 Automatic installation

The `RAppArmor` package consists of `R` software and a number of example security profiles. On Ubuntu it is easiest installed using binary builds provided through launchpad:

```
sudo add-apt-repository ppa:opencpu/rapparmor
sudo apt-get update
sudo apt-get install r-cran-rapparmor
```

Binaries in this repository are build for the version of `R` that ships with the operating system. To get builds for `R` version 3.0 and up, use repository `ppa:opencpu/rapparmor-dev` instead. The `r-cran-rapparmor` package can also be build from source using something along the lines of the following:

```
sudo apt-get install libapparmor-dev devscripts
wget http://cran.fhcrc.org/src/contrib/Archive/RAppArmor/RAppArmor_1.0.0.tar.gz
tar xzvf RAppArmor_1.0.0.tar.gz
cd RAppArmor/
debuild -uc -us
cd ..
sudo dpkg -i r-cran-rapparmor_*.deb
```

The `r-cran-rapparmor` package will automatically install required dependencies and security profiles. The security profiles are installed in the directory `/etc/appamor.d/rapparmor.d/`.

3.4.3 Manual installation

On distributions for which no system installation package is available, manual installation is required. Start with installing required dependencies:

```
sudo apt-get install r-base-dev libapparmor-dev apparmor apparmor-utils
```

Note that `R` version 2.14 or higher is required. Also the system needs to have an `AppArmor` enabled `Linux` kernel. After dependencies have been installed, install `RAppArmor` from CRAN:

```
wget http://cran.r-project.org/src/contrib/RAppArmor_1.0.0.tar.gz
sudo R CMD INSTALL RAppArmor_1.0.0.tar.gz
```

This will compile the `C` code and install the `R` package. After the package has been installed successfully, the security profiles need to be copied to the `apparmor.d` directory:

```
cd /usr/local/lib/R/site-library/RAppArmor/
sudo cp -Rf profiles/debian/* /etc/apparmor.d/
```

Finally, the `AppArmor` service needs to be restarted to load the new profiles. Also we do not want to enforce the global `R` profile at this point:

```
sudo service apparmor restart
sudo aa-disable usr.bin.r
```

This should complete the installation. To verify if everything is working, start `R` and run the following code:

```
library("RAppArmor")

## AppArmor LSM is enabled.
## Current profile:  none (unconfined).
## See ?aa_change_profile on how switch profiles.

aa_change_profile("r-base")

## Switching profiles...
```

If the code runs without any errors, the package has successfully been installed.

3.4.4 Linux security methods

The `RAppArmor` package interfaces to a number of `Linux` system calls that are useful in the context of security and sandboxing. The advantage of calling these directly from `R` is that we can dynamically set the parameters from within the `R` process, as opposed to fixing them for all `R` sessions. Hence it is actually possible to execute some parts of an application in a different security context other parts.

The package implements many functions that wrap around `Linux C` interfaces. However it is not required to study all of these functions. To the end user, everything in the package comes together in the powerful and convenient `eval.secure()` function. This function mimics `eval()`, but it has additional parameters that define restrictions which will be enforced to a specific evaluation. An example:

```
myresult <- eval.secure(myfun(), RLIMIT_AS = 1024 * 1024, profile = "r-base")
```

This will call `myfun()` with a memory limit of 10MB and the "r-base" security profile (which is introduced in section 3.5.2). The `eval.secure` function works by creating a *fork* of the current process, and then sets hard limits, `uid` and `AppArmor` profile on the forked process, before evaluating the call. After the function returns, or when the timeout is reached, the forked process is killed and cleaned up. This way, all of the one-way security restrictions can be applied, and evaluations that happen inside `eval.secure` won't have any side effects on the main process.

3.4.5 Setting user and group ID

One of the most basic security methods is running a process as a specific user. Especially within a system where the main process has superuser privileges (which could be the case in for example a webserver), switching to a user with limited privileges before evaluating any code is a wise thing to do. We could even consider a design where every user of the application has a dedicated user account on the `Linux` machine. The `RAppArmor` package implements the functions `getuid, setuid, getgid, setgid`, which call out to the respective `Linux` system calls. Users and groups can either be specified by their name, or as integer values as defined in the `/etc/passwd` file.

```
# run as root
system("whoami", intern = TRUE)

## [1] "root"

getuid()

## [1] 0

getgid()

## [1] 0

setgid(1000)
setuid(1000)
getgid()

## [1] 1000
```

```r
getuid()

## [1] 1000

system("whoami", intern = TRUE)

## [1] "jeroen"
```

The user/group ID can also be set inside the **eval.secure** function. In this case it will not affect the main process; the UID is only set for the time of the secure evaluation.

```r
# run as root
eval(system("whoami", intern = TRUE))

## [1] "root"

eval.secure(system("whoami", intern = TRUE), uid = 1000)

## [1] "jeroen"

eval(system("whoami", intern = TRUE))

## [1] "root"
```

Note that in order for **setgid** and **setuid** to work, the user must have the appropriate capabilities in **Linux**, which are usually restricted to users with superuser privileges. The **getuid** and **getgid** functions can be called by anyone.

3.4.6 Setting Task Priority

The **RAppArmor** package implements interfaces for setting the scheduling priority of a process, also called its **nice** value or *niceness*. **Linux** systems use a priority system with 40 priorities, ranging from -20 (highest priority) to 19 (lowest priority). By default most processes run with nice value 0. Users without superuser privileges can only increase this value, i.e. lower the priority of the process. In `RAppArmor` the `getpriority` and `setpriority` functions change the priority of the current session:

```
getpriority()

## [1] 0

setpriority(5)

## [1] 5

system("nice", intern = TRUE)

## [1] "5"

setpriority(0)

## Error:  Failed to set priority.
```

Again, the `eval.secure` function is used to run a function or code block with a certain priority without affecting the priority of the main `R` session:

```
getpriority()

## [1] 5

eval.secure(system("nice", intern = TRUE), priority = 10)

## [1] "10"

getpriority()

## [1] 5
```

3.4.7 Linux Resource Limits (RLIMIT)

Linux defines a number of `RLIMIT` values that can be used to set resource limits on a process (Free Software Foundation, 2012). The `RAppArmor` package has functions to get/set to the following RLIMITs:

- `RLIMIT_AS` – The maximum size of the process's virtual memory (address space).
- `RLIMIT_CORE` – Maximum size of core file.
- `RLIMIT_CPU` – CPU time limit.
- `RLIMIT_DATA` – The maximum size of the process's data segment.
- `RLIMIT_FSIZE` – The maximum size of files that the process may create.
- `RLIMIT_MEMLOCK` – Number of memory that may be locked into RAM.
- `RLIMIT_MSGQUEUE` – Max number of bytes that can be allocated for POSIX message queues
- `RLIMIT_NICE` – Specifies a ceiling to which the process's nice value (priority).
- `RLIMIT_NOFILE` – Limit maximum file descriptor number that can be opened.
- `RLIMIT_NPROC` – Maximum number of processes (or, more precisely on Linux, threads) that can be created by the user of the calling process.
- `RLIMIT_RTPRIO` – Ceiling on the real-time priority that may be set for this process.
- `RLIMIT_RTTIME` – Limit on the amount of CPU time that a process scheduled under a real-time scheduling policy may consume without making a blocking system call.

- RLIMIT_SIGPENDING – Limit on the number of signals that may be queued by the user of the calling process.
- RLIMIT_STACK – The maximum size of the process stack.

For all of the above RLIMITs, the RAppArmor package implements a function which name is equivalent to the non-capitalized name of the RLIMIT. For example to get/set RLIMIT_AS, the user calls rlimit_as(). Every rlimit_ function has exactly 3 parameters: hardlim, softlim, and pid. Each argument is specified as an integer value. The pid arguments points to the target process. When this argument is omitted, the calling process is targeted. When the softlim is omitted, it is set equal to the hardlim. When the function is called without any arguments, it returns the current limits.

The soft limit is the value that the kernel enforces for the corresponding resource. The hard limit acts as a ceiling for the soft limit: an unprivileged process may only set its soft limit to a value in the range from 0 up to the hard limit, and (irreversibly) lower its hard limit. A privileged process (under Linux: one with the CAP_SYS_RESOURCE capability) may make arbitrary changes to either limit value. (Free Software Foundation, 2012)

```
A <- rnorm(1e+07)
rm(A)
gc()

##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells 377297 20.2     667722 35.7   597831 32.0
## Vcells 548431  4.2    9056788 69.1 10562454 80.6

rlimit_as(10 * 1024 * 1024)
A <- rnorm(1e+07)

## Error:  cannot allocate vector of size 76.3 Mb
```

Note that a process owned by a user without superuser privileges can only modify `RLIMIT` to more restrictive values. However, using `eval.secure`, a more restrictive `RLIMIT` can be applied to a single evaluation without any side effects on the main process:

```
A <- eval.secure(rnorm(1e+07), RLIMIT_AS = 10 * 1024 * 1024)

## Error:  cannot allocate vector of size 76.3 Mb

A <- rnorm(1e+07)
object.size(A)

## 80000040 bytes
```

The exact meaning of the different limits can be found in the `RAppArmor` package documentation (e.g. `?rlimit_as`) or in the documentation of the distribution, e.g. Canonical, Inc (2012).

3.4.8 Activating `AppArmor` profiles

The `RAppArmor` package implements three calls to the `Linux` kernel related to applying `AppArmor` profiles: `aa_change_profile`, `aa_change_hat` and `aa_revert_hat`. Both the `aa_change_profile` and `aa_change_hat` functions take a parameter named `profile`: a character string identifying the name of the profile. This profile has to be preloaded by the kernel, before it can be applied to a process. The easiest way to load profiles is to copy them to the directory `/etc/apparmor.d/` and then run `sudo service apparmor restart`.

The main difference between a *profile* and a *hat* is that switching profiles is an irreversible action. Once the profile has been associated with the current process, the process cannot call `aa_change_profile` again to escape from the profile (that would defeat the purpose). The only exception to this rule are profiles that contain an explicit `change_profile` directive. The `aa_change_hat` function on the other hand is designed to associate a process with a security profile in a way that does allow it to escape out of the security profile. In order to realize this, the `aa_change_hat` takes a second argument called `magic_token`, which defines a secret key that can be used to *revert* the hat. When `aa_revert_hat` is called with the same `magic_token` that was used in `aa_change_hat`, the security restrictions are relieved.

Using `aa_change_hat` to switch in and out of profiles is an easy way to get started with `RAppArmor` and test some security policies. However it should be emphasized that using *hats* instead of *profiles* is also a security risk and should be avoided in production settings. It is important to realize that if the code running in the sandbox can find a way of discovering the value of the `magic_token` (e.g. from memory, command history or log files), it will be able to escape from the sandbox. Hence `aa_change_hat` should only be used to prevent general purpose malicious activity, e.g. when testing a new `R` package. When hosting services or otherwise exposing an environment that might be specifically targeted, hackers could write code that attempts to find the magic token and revert the hat. Therefore it is recommended to only use `aa_change_profile` or `eval.secure` in production settings. When a profile is applied to a process using `aa_change_profile` or `eval.secure`, the kernel will keep enforcing the security policy on the respective process and all of its children until they die, no matter what.

The `RAppArmor` package ships with a profile called *testprofile* which contains a hat called *testhat*. We use this profile to demonstrate the functionality. The profiles have been defined such that *testprofile* allows access to `/etc/group` but denies access to `/etc/passwd`. The *testhat* denies access to both `/etc/passwd` and `/etc/group`.

```
aa_getcon()$con

## Getting task confinement information...
## [1] "unconfined"

result <- read.table("/etc/passwd")

# load the profile
aa_change_profile("testprofile")

## Switching profiles...

aa_getcon()$con

## Getting task confinement information...
## [1] "testprofile"

passwd <- read.table("/etc/passwd")

## Warning:  cannot open file '/etc/passwd':  Permission denied
## Error:  cannot open the connection

group <- read.table("/etc/group")

# switch into the 'hat'
mytoken <- 13337
aa_change_hat("testhat", mytoken)

## Setting AppArmor Hat...

aa_getcon()$con

## Getting task confinement information...
## [1] "testprofile//testhat"

passwd <- read.table("/etc/passwd")
```

```
## Warning:  cannot open file '/etc/passwd':  Permission denied

## Error:  cannot open the connection

group <- read.table("/etc/group")

## Warning:  cannot open file '/etc/group':  Permission denied

## Error:  cannot open the connection

# revert the 'hat'
aa_revert_hat(mytoken)

## Reverting AppArmor Hat...

aa_getcon()$con

## Getting task confinement information...
## [1] "testprofile"

passwd <- read.table("/etc/passwd")

## Warning:  cannot open file '/etc/passwd':  Permission denied

## Error:  cannot open the connection

group <- read.table("/etc/group")
```

Just like for `setuid` and `rlimit` functions, `eval.secure` can be used to enforce an `AppArmor` security profile on a single call, witout any side effects. The `eval.secure` function uses `aa_change_profile` and is therefore most secure.

```
out <- eval(read.table("/etc/passwd"))
nrow(out)

## [1] 66

out <- eval.secure(read.table("/etc/passwd"), profile = "testprofile")
```

3.4.9 AppArmor without RAppArmor

The `RAppArmor` package allows us to dynamically load an `AppArmor` profile from within an `R` session. This gives a great deal of flexibility. However, it is also possible to use `AppArmor` without the `RAppArmor` package, by setting a single profile to be loaded with any running `R` process.

To do so, the RAppArmor package ships with a profile named `usr.bin.r`. At the installation of the package, this file is copied to `/etc/apparmor.d/`. This file is basically a copy of the `r-user` profile in appendix A.3, however with a small change: where `r-user` defines a named profile with

```
profile r-user {
  ...
}
```

the `usr.bin.r` file defines a profile specific to a filepath:

```
/usr/bin/R {
  ...
 }
```

When using the latter syntax, the profile is automatically associated every time the file `/usr/bin/R` is executed (which is the script that runs when `R` is started from the shell). This way we can set some default security restrictions for our daily work. Profiles tied to a specific program can be activated only by the administrator using:

```
sudo aa-enforce usr.bin.r
```

This will enforce the security restrictions on every new `R` process that is started. To stop enforcing the restrictions, the administrator can run:

```
sudo aa-disable usr.bin.r
```

After disabling the profile, the `R` program can be started without any restrictions. Note that the `usr.bin.r` profile does **not** grant permission to change profiles. Hence, once the `usr.bin.r` profile is in enforce mode, we cannot use the `eval.secure` or `aa_change_profile` functions from the `RAppArmor` package to change into a different profile, as this would be a security hole:

```
library(RAppArmor)

## AppArmor LSM is enabled.
## Current profile: /usr/bin/R (enforce mode)

aa_change_profile("r-user")

## Switching profiles...
## Getting task confinement information...

## Warning: The standard profile in usr.bin.r is already being enforced!
##  Run sudo aa-disable usr.bin.r to disable this.
## Error: Failed to change profile from: /usr/bin/R to: r-user.
```

3.4.10 Learning using complain mode

Finally `AppArmor` allows the administrator to set profiles in *complain mode*, which is also called *learning mode*.

```
sudo aa-complain usr.bin.r
```

This is useful for developing new profiles. When a profile is set in complain mode, security restrictions are not actually enforced; instead all violations of the security policy are logged to the `syslog` and `kern.log` files. This is a powerful way of creating new profiles: a program can be set in complain mode during regular use, and afterwards the log files can be used to study violations of the current policy. From these violations we can determine which permissions need to be added to the profile to make the program work under normal behavior. `AppArmor` even ships with a utility named `aa-logprof` which can help the administrator by parsing these log files and suggesting new rules to be added to the profile. This is a nice way of debugging a profile, and figure out which permissions exactly a program requires to do its work.

3.5 Profiling `R`: defining security policies

The "hard" part of the problem is actually profiling `R`. With profiling we mean defining the policies: which files and directories should `R` be allowed to read and write to? Which external programs is it allowed to execute? Which libraries or shared modules it allowed to load, etc. We want to minimize ways in which the process could potentially damage the system, but we don't want to be overly restrictive either: preferebly, users should be able to do anything they normally do in `R`. Because `R` is such a complete system with a big codebase and a wide range of functionality, the base system actually already requires quite a lot of access to the file system.

As often, there is no "one size fits all" solution. Depending on which functionality is needed for an application we might want to grant or deny certain privileges. We might even want to execute some parts of a process with tighter privileges than other parts. For example, within a web service, the service process should be able to write to system log files, which should not be writable by custom code from a user. We might also want to be more strict on some users than others, e.g. allow all users to run code, but only allow privileged users to install a new package.

3.5.1 `AppArmor` policy configuration syntax

The *AppArmor policy configuration syntax* is used to define the access control profiles in `AppArmor`. Other mandatory access control systems might implement different functionality and require other syntax, but in the end they address mostly similar issues. `AppArmor` is quite advanced and provides access control over many of the features and resources found in the `Linux` kernel, e.g. file access, network rules, `Linux` capability modes, mounting rules, etc. All of these can be useful, but most of them are very application specific. Furthermore, the policy syntax has some meta functionality that allows for defining *subprofiles*, and *includes*.

The most important form of access control which will be the focus of the remaining of the section are *file permission access modes*. Once `AppArmor` is enforcing mandatory access control, a process can only access files and directories on the system for which it has explicitly been granted access in its security profile. Because in `Linux` almost everything is a file (even sockets, devices, etc) this gives a great deal of control. `AppArmor` defines a number of access modes on files and directories, of which the most important ones are:

`r` – read file or directory.

`w` – write to file or directory.

`m` – load file in memory.

`px` – discrete profile execute of executable file.

`cs` – transition to subprofile for executing a file.

`ix` – inherit current profile for executing a file.

`ux` – unconfined execution of executable file (dangerous).

Using this syntax we will present some example profiles for R. Because the profiles are defined using absolute paths of system files, we will assume the standard file layout for Debian and Ubuntu systems. This includes files that are part of r-base and other packages that are used by R, e.g. texlive, libxml2, bash, libpango, libcairo, etc.

3.5.2 Profile: r-base

Appendix A.1 contains a profile that we have named r-base. It is a fairly basic and general profile. It grants read/load access to all files in common shared system directories, e.g. /usr/lib, /usr/local/lib, /usr/share, etc. However, the default profile only grants write access inside /tmp, not in e.g. the home directory. Furthermore, R is allowed to execute any of the shell commands in /bin or /usr/bin for which the program will inherit the current restrictions.

```
# base profile
aa_change_profile("r-base")

## Switching profiles...
```

This profile denies access to most systems files and the home directory:

```
# stuff that is not allowed
list.files("/")

## character(0)

list.files("~")

## character(0)

file.create("~/test")

## Warning:  cannot create file '~/test', reason 'Permission denied'
## [1] FALSE

list.files("/tmp")

## character(0)
```

However the profile does grant access to the global library and temporary directory:

```
# stuff that is allowed
library(MASS)
setwd(tempdir())
pdf("test.pdf")
plot(speed ~ dist, data = cars)
dev.off()

## pdf
##   2

list.files()

## [1] "test.pdf"

file.remove("test.pdf")

## [1] TRUE
```

The `r-base` profile effectively protects `R` from most malicious activity, while still allowing access to all of the libraries, fonts, icons, and programs that it might need. One thing to note is that the profile does not allow listing of the contents of `/tmp`, but it does allow full `rw` access on any of its subdirectories. This is to prevent one process from reading/writing files in the temp directory of another active `R` process (given that it cannot discover the name of the other temp directory).

The `r-base` profile is a quite liberal and general purpose profile. When using `AppArmor` in a more specific application, it is recommended to make the profile a bit more restrictive by specifying exactly *which* of the packages, shell commands and system libraries should be accessible by the application. That could prevent potential problems when vulnerabilities are discovered in some of the standard libraries.

3.5.3 Profile: `r-compile`

The `r-base` profile does not allow access to the compiler, nor does it allow for loading (`m`) or execution (`ix`) of files in places where it can also write. If we want user to be able to compile e.g. `C++` code, the policy needs grant access to the compiler. Assuming `GCC` is installed, the following lines can be added to the profile:

```
/usr/include/** r,
/usr/lib/gcc/** rix,
/tmp/** rmw,
```

Note especially the last line. The combination of `w` and `m` access modes allows `R` to load a shared object into memory from after installing it in a temporary directory. This does not come without a cost: compiled code can potentially contain malicious code or even exploits that can do harm when loaded into memory. If this privilege is not needed, it is generally recommended to only allow `m` and ix access modes on files that have been installed by the system administrator. The new profile including these rules ships with the package as `r-compile` and is also printed in appendix A.2.

After adding the lines above and reloading the profile, it should be possible to compile a package that contains `C++` code and install it to somewhere in `/tmp`:

```
eval.secure(install.packages("wordcloud", lib = tempdir()),
  profile = "r-compile")
```

3.5.4 Profile: `r-user`

Appendix A.3 defines a profile named `r-user`. This profile is designed to be a nice balance between security and freedom for day to day use of R. It extends the `r-compile` profile with some additional privileges in the user's home directory. The variable `@{HOME}` is defined in the `/etc/apparmor.d/tunables/global` include that ships with `AppArmor` and matches the location of the user home directory, i.e. `/home/jeroen/`. If a directory named `R` exists inside the home directory (e.g `/home/jeroen/R/`), `R` has both read and write permissions here. Furthermore, `R` can load and execute files in the directories `i686-pc-linux-gnu-library` and `x86_64-pc-linux-gnu-library` inside of this directory. These are the standard locations where `R` installs a user's personal package library.

With the `r-user` profile, we can do most of our day to day work, including installing and loading new packages in our personal library, while still being protected against most malicious activities. The `r-user` profile is also the basis of the default `usr.bin.r` profile mentioned in section 3.4.9.

3.5.5 Installing packages

An additional privilege that might be needed in some situations is the option to install packages to the system's global library, which is readable by all users. In order to allow this, a profile needs to include write access to the `site-library` directory:

```
/usr/local/lib/R/site-library/ rw,
/usr/local/lib/R/site-library/** rwm,
```

With this rule, the policy will allow for installing `R` packages to the global site library. However, note that `AppArmor` does not replace, but *supplements* the standard access control system. Hence if a user does not have permission to write into this directory (either by standard Unix access controls or by running with superuser privileges), it will still not be able to install packages in the global site library, even though the `AppArmor` profile does grant this permission.

3.6 Concluding remarks

In this paper the reader was introduced to some potential security issues related to the use of the `R` software. We hope to have raised awareness that security is an increasingly important concern for the `R` user, but also that addressing this issue could open doors to new applications of the software. The `RAppArmor` package was introduced as an example that demonstrates how some security issues could be addressed using facilities from the operating system, in this case `Linux`. This implementation provides a starting point for creating a sandbox in `R`, but as was emphasized throughout the paper, it is still up to the administrator to actually design security policies that are appropriate for a certain application or system.

Our package uses the `AppArmor` software from the `Linux` kernel, which works for us, but this is just one of the available options. `Linux` has two other mandatory access control systems that are worth exploring: `TOMOYO` and `SELinux`. Especially the latter is known to be very sophisticated, but also extremely hard to set up. Other technology that might be interesting is provided by `Linux CGroups`. Using `CGroups`, control of allocation and security is managed by hierarchical process groups. The more recent `LXC` (Linux Containers) build on `CGroups` to provide virtual environments which have their own process and network space. A completely different direction is suggested by `renjin` (Bertram, 2012), a `JVM`-based interpreter for the R Language. If `R` code can be executed though the `JVM`, we might be able to use some tools from the `Java` community to address similar issues. Finally the `TrustedBSD` project provides advanced security features which could provide a foundation for sandboxing `R` on `BSD` systems.

However, regardless of the tools that are used, security always comes down to the trade off between *use* and *abuse*. This has a major human aspect to it, and is a learning process in itself. A balance has to be found between providing enough freedom to use facilities as desired, yet minimize opportunities for undesired activity. Apart from technical parameters, a good balance also depends on factors like what exactly constitutes undesired behavior and the relation between users and provider. For example a process using 20 parallel cores might be considered abusive by some administrators, but might actually be regular use for a MCMC simulation server. Security policies are not unlike legal policies in the sense that they won't always immediately work out as intended, and need to evolve over time as part of an iterative process. It might not be until an application is put in production that users start complaining about their favorite package not working, or that we find the system being abused in a way that was hard to foresee. We hope that our research will contribute to this process and help take a step in the direction of a safer `R`.

CHAPTER 4

The `jsonlite` Package: A Practical and Consistent Mapping Between `JSON` Data and `R` Objects

4.1 Introduction

JavaScript Object Notation (`JSON`) is a text format for the serialization of structured data (Crockford, 2006a). It is derived from the object literals of `JavaScript`, as defined in the `ECMAScript` programming language standard (Ecma International, 1999). Design of `JSON` is simple and concise in comparison with other text based formats, and it was originally proposed by Douglas Crockford as a "fat-free alternative to `XML`" (Crockford, 2006b). The syntax is easy for humans to read and write, easy for machines to parse and generate and completely described in a single page at `http://www.json.org`. The character encoding of `JSON` text is always Unicode, using `UTF-8` by default (Crockford, 2006a), making it naturally compatible with non-latin alphabets. Over the past years, `JSON` has become hugely popular on the internet as a general purpose data interchange format. High quality parsing libraries are available for almost any programming language, making it easy to implement systems and applications that exchange data over the network using `JSON`. For `R` (R Core Team, 2014a), several packages that assist the user in generating, parsing and validating `JSON` are available through CRAN, including `rjson` (Couture-Beil, 2013), `RJSONIO` (Lang, 2012), and `jsonlite` (Ooms et al., 2014).

The emphasis of this paper is not on discussing the `JSON` format or any particular implementation for using `JSON` with `R`. We refer to Nolan and Temple Lang (2014) for a comprehensive introduction, or one of the many tutorials available on the web. Instead we take a high level view and discuss how `R` data structures are most naturally represented in `JSON`. This is not a trivial problem, particularly for complex or relational data as they frequently appear in statistical applications. Several `R` packages implement `toJSON` and `fromJSON` functions which directly convert `R` objects into `JSON` and vice versa. However, the exact mapping between the various `R` data classes `JSON` structures is not self evident. Currently, there are no formal guidelines, or even consensus between implementations on how `R` data should be represented in `JSON`. Furthermore, upon closer inspection, even the most basic data structures in `R` actually do not perfectly map to their `JSON` counterparts, and leave some ambiguity for edge cases. These problems have resulted in different behavior between implementations, and can lead to unexpected output for certain special cases. Furthermore, best practices of representing data in `JSON` have been established outside the `R` community. Incorporating these conventions where possible is important to maximize interoperability.

4.1.1 Parsing and type safety

The `JSON` format specifies 4 primitive types (`string`, `number`, `boolean`, `null`) and two *universal structures*:

- A `JSON` *object*: an unordered collection of zero or more name-value pairs, where a name is a string and a value is a string, number, boolean, null, object, or array.
- A `JSON` *array*: an ordered sequence of zero or more values.

Both these structures are heterogeneous; i.e. they are allowed to contain elements of different types. Therefore, the native `R` realization of these structures is a `named list` for `JSON` objects, and `unnamed list` for `JSON` arrays. However, in practice a list is an awkward, inefficient type to store and manipulate data in `R`. Most statistical applications work with (homogeneous) vectors, matrices or data frames. In order to give these data structures a `JSON` representation, we can define certain special cases of `JSON` structures which get parsed into other, more specific `R` types. For example, one convention which all current implementations have in common is that a homogeneous array of primitives gets parsed into an `atomic vector` instead of a `list`. The `RJSONIO` documentation uses the term "simplify" for this behavior, and we adopt this jargon.

```
txt <- "[12, 3, 7]"
x <- fromJSON(txt)
is(x)

[1] "numeric" "vector"

print(x)

[1] 12  3  7
```

This seems very reasonable and it is the only practical solution to represent vectors in `JSON`. However the price we pay is that automatic simplification can compromise type-safety in the context of dynamic data. For example, suppose an `R` package uses `fromJSON` to pull data from a `JSON API` on the web and that for some particular combination of parameters the result includes a `null` value, e.g: `[12, null, 7]`. This is actually quite common, many `API`'s use `null` for missing values or unset fields. This case makes the behavior of parser ambiguous, because the `JSON` array is technically no longer homogeneous. And indeed, some implementations will now return a `list` instead of a `vector`. If the user had not anticipated this scenario and the script assumes a `vector`, the code is likely to run into type errors.

The lesson here is that we need to be very specific and explicit about the mapping that is implemented to convert between JSON data and R objects. When relying on JSON as a data interchange format, the behavior of the parser must be consistent and unambiguous. Clients relying on JSON to get data in and out of R must know exactly what to expect in order to facilitate reliable communication, even if the content of the data is dynamic. Similarly, R code using dynamic JSON data from an external source is only reliable when the conversion from JSON to R is consistent. Moreover a practical mapping must incorporate existing conventions and use the most natural representation of certain structures in R. In the example above, we could argue that instead of falling back on a `list`, the array is more naturally interpreted as a numeric vector where the `null` becomes a missing value (NA). These principles will extrapolate as we start discussing more complex JSON structures representing matrices and data frames.

4.1.2 Reference implementation: the `jsonlite` package

The `jsonlite` package provides a reference implementation of the conventions proposed in this document. It is a fork of the RJSONIO package by Duncan Temple Lang, which builds on `libjson` C++ library from Jonathan Wallace. `jsonlite` uses the parser from RJSONIO, but the R code has been rewritten from scratch. Both packages implement `toJSON` and `fromJSON` functions, but their output is quite different. Finally, the `jsonlite` package contains a large set of unit tests to validate that R objects are correctly converted to JSON and vice versa. These unit tests cover all classes and edge cases mentioned in this document, and could be used to validate if other implementations follow the same conventions.

```
library(testthat)
test_package("jsonlite")
```

Note that even though JSON allows for inserting arbitrary white space and indentation, the unit tests assume that white space is trimmed.

4.1.3 Class-based versus type-based encoding

The `jsonlite` package actually implements two systems for translating between `R` objects and `JSON`. This document focuses on the `toJSON` and `fromJSON` functions which use `R`'s class-based method dispatch. For all of the common classes in `R`, the `jsonlite` package implements `toJSON` methods as described in this document. Users in `R` can extend this system by implementing additional methods for other classes. This also means that classes that do not have the `toJSON` method defined are not supported. Furthermore, the implementation of a specific `toJSON` method determines which data and metadata in the objects of this class gets encoded in its `JSON` representation, and how. In this respect, `toJSON` is similar to e.g. the `print` function, which also provides a certain *representation* of an object based on its class and optionally some print parameters. This representation does not necessarily reflect all information stored in the object, and there is no guaranteed one-to-one correspondence between `R` objects and `JSON`. I.e. calling `fromJSON(toJSON(object))` will return an object which only contains the data that was encoded by the `toJSON` method for this particular class, and which might even have a different class than the original.

The alternative to class-based method dispatch is to use type-based encoding, which `jsonlite` implements in the functions `serializeJSON` and `unserializeJSON`. All data structures in `R` get stored in memory using one of the internal `SEXP` storage types, and `serializeJSON` defines an encoding schema which captures the type, value, and attributes for each storage type. The resulting `JSON` closely resembles the internal structure of the underlying `C` data types, and can be perfectly restored to the original `R` object using `unserializeJSON`. This system is relatively straightforward to implement, but the resulting `JSON` is very verbose, hard to interpret, and cumbersome to generate in the context of another language or system. For most applications this is actually impractical because it requires the client/consumer to understand and manipulate R data types, which is difficult and reduces interoperability. Instead we can make data in R more accessible to third parties by defining sensible JSON representations that are natural for the class of an object, rather than its internal storage type. This document does not discuss the serializeJSON system in any further detail, and solely treats the class based system implemented in toJSON and fromJSON. However the reader that is interested in full serialization of R objects into JSON is encouraged to have a look at the respective manual pages.

4.1.4 Scope and limitations

Before continuing, we want to stress some limitations of encoding R data structures in JSON. Most importantly, there are limitations to the types of objects that can be represented. In general, temporary in-memory properties such as connections, file descriptors and (recursive) memory references are always difficult if not impossible to store in a sensible way, regardless of the language or serialization method. This document focuses on the common R classes that hold *data*, such as vectors, factors, lists, matrices and data frames. We do not treat language level constructs such as expressions, functions, promises, which hold little meaning outside the context of R. We also don't treat special compound classes such as linear models or custom classes defined in contributed packages. When designing systems or protocols that interact with R, it is highly recommended to stick with the standard data structures for the interface input/output.

Then there are limitations introduced by the format. Because JSON is a human readable, text-based format, it does not support binary data, and numbers are stored in their decimal notation. The latter leads to loss of precision for real numbers, depending on how many digits the user decides to print. Several dialects of JSON exists such as BSON (Chodorow, 2013) or MSGPACK (Furuhashi, 2014), which extend the format with various binary types. However, these formats are much less popular, less interoperable, and often impractical, precisely because they require binary parsing and abandon human readability. The simplicity of JSON is what makes it an accessible and widely applicable data interchange format. In cases where it is really needed to include some binary data in JSON, we can encode a blob as a string using base64.

Finally, as mentioned earlier, fromJSON is not a perfect inverse function of toJSON, as is the case for serialializeJSON and unserializeJSON. The class based mappings are designed for concise and practical encoding of the various common data structures. Our implementation of toJSON and fromJSON approximates a reversible mapping between R objects and JSON for the standard data classes, but there are always limitations and edge cases. For example, the JSON representation of an empty vector, empty list or empty data frame are all the same: "[ ]". Also some special vector types such as factors, dates or timestamps get coerced to strings, as they would in for example CSV. This is a quite typical and expected behavior among text based formats, but it does require some additional interpretation on the consumer side.

4.2 Converting between JSON and R classes

This section lists examples of how the common R classes are represented in JSON. As explained before, the toJSON function relies on method dispatch, which means that objects get encoded according to their class attribute. If an object has multiple class values, R uses the first occurring class which has a toJSON method. If none of the classes of an object has a toJSON method, an error is raised.

4.2.1 Atomic vectors

The most basic data type in `R` is the atomic vector. Atomic vectors hold an ordered, homogeneous set of values of type `"logical"` (booleans), `character` (strings), `"raw"` (bytes), `numeric` (doubles), `"complex"` (complex numbers with a real and imaginary part), or `integer`. Because `R` is fully vectorized, there is no user level notion of a primitive: a scalar value is considered a vector of length 1. Atomic vectors map to `JSON` arrays:

```
x <- c(1, 2, pi)
cat(toJSON(x))

[ 1, 2, 3.14 ]
```

The `JSON` array is the only appropriate structure to encode a vector, even though vectors in `R` are homogeneous, whereas the `JSON` array is actually heterogeneous, but `JSON` does not make this distinction.

4.2.1.1 Missing values

A typical domain specific problem when working with statistical data is presented by missing values: a concept foreign to many other languages. Besides regular values, each vector type in `R` except for `raw` can hold `NA` as a value. Vectors of type `double` and `complex` define three additional types of non finite values: `NaN`, `Inf` and `-Inf`. The `JSON` format does not natively support any of these types; therefore such values values need to be encoded in some other way. There are two obvious approaches. The first one is to use the `JSON` `null` type. For example:

```
x <- c(TRUE, FALSE, NA)
cat(toJSON(x))

[ true, false, null ]
```

The other option is to encode missing values as strings by wrapping them in double quotes:

```
x <- c(1, 2, NA, NaN, Inf, 10)
cat(toJSON(x))

[ 1, 2, "NA", "NaN", "Inf", 10 ]
```

Both methods result in valid `JSON`, but both have a limitation: the problem with the `null` type is that it is impossible to distinguish between different types of missing data, which could be a problem for numeric vectors. The values `Inf`, `-Inf`, `NA` and `NaN` carry different meanings, and these should not get lost in the encoding. The problem with encoding missing values as strings is that this method can not be used for character vectors, because the consumer won't be able to distinguish the actual string `"NA"` and the missing value `NA`. This would create a likely source of bugs, where clients mistakenly interpret `"NA"` as an actual string value, which is a common problem with text-based formats such as `CSV`. For this reason, `jsonlite` uses the following defaults:

- Missing values in non-numeric vectors (`logical`, `character`) are encoded as `null`.
- Missing values in numeric vectors (`double`, `integer`, `complex`) are encoded as strings.

We expect that these conventions are most likely to result in the correct interpretation of missing values. Some examples:

```
cat(toJSON(c(TRUE, NA, NA, FALSE)))

[ true, null, null, false ]

cat(toJSON(c("FOO", "BAR", NA, "NA")))
```

```
[ "FOO", "BAR", null, "NA" ]

cat(toJSON(c(3.14, NA, NaN, 21, Inf, -Inf)))

[ 3.14, "NA", "NaN", 21, "Inf", "-Inf" ]

# We can override default behavior
cat(toJSON(c(3.14, NA, NaN, 21, Inf, -Inf), na = "null"))

[ 3.14, null, null, 21, null, null ]
```

4.2.1.2 Special vector types: dates, times, factor, complex

Besides missing values, `JSON` also lacks native support for some of the basic vector types in `R` that frequently appear in data sets. These include vectors of class `Date`, `POSIXt` (timestamps), `factors` and `complex` vectors. By default, the `jsonlite` package coerces these types to strings (using `as.character`):

```
cat(toJSON(Sys.time() + 1:3))

[ "2014-05-11 17:27:03", "2014-05-11 17:27:04", "2014-05-11 17:27:05" ]

cat(toJSON(as.Date(Sys.time()) + 1:3))

[ "2014-05-13", "2014-05-14", "2014-05-15" ]

cat(toJSON(factor(c("foo", "bar", "foo"))))

[ "foo", "bar", "foo" ]

cat(toJSON(complex(real = runif(3), imaginary = rnorm(3))))

[ "0.4-2.6i", "0.61+0.05i", "0.6+1.9i" ]
```

When parsing such JSON strings, these values will appear as character vectors. In order to obtain the original types, the user needs to manually coerce them back to the desired type using the corresponding as function, e.g. as.POSIXct, as.Date, as.factor or as.complex. In this respect, JSON is subject to the same limitations as text based formats such as CSV.

4.2.1.3 Special cases: vectors of length 0 or 1

Two edge cases deserve special attention: vectors of length 0 and vectors of length 1. In jsonlite these are encoded respectively as an empty array, and an array of length 1:

```
# vectors of length 0 and 1
cat(toJSON(vector()))

[  ]

cat(toJSON(pi))

[ 3.14 ]

# vectors of length 0 and 1 in a named list
cat(toJSON(list(foo = vector())))

{ "foo" : [  ] }

cat(toJSON(list(foo = pi)))

{ "foo" : [ 3.14 ] }

# vectors of length 0 and 1 in an unnamed list
cat(toJSON(list(vector())))

[ [  ] ]

cat(toJSON(list(pi)))

[ [ 3.14 ] ]
```

This might seem obvious but these cases result in very different behavior between different JSON packages. This is probably caused by the fact that R does not have a scalar type, and some package authors decided to treat vectors of length 1 as if they were a scalar. For example, in the current implementations, both RJSONIO and rjson encode a vector of length one as a JSON primitive when it appears within a list:

```
# Other packages make different choices:
cat(rjson::toJSON(list(n = c(1))))

 {"n":1}

cat(rjson::toJSON(list(n = c(1, 2))))

 {"n":[1,2]}
```

When encoding a single dataset this seems harmless, but in the context of dynamic data this inconsistency is almost guaranteed to cause bugs. For example, imagine an R web service which lets the user fit a linear model and sends back the fitted parameter estimates as a JSON array. The client code then parses the JSON, and iterates over the array of coefficients to display them in a GUI. All goes well, until the user decides to fit a model with only one predictor. If the JSON encoder suddenly returns a primitive value where the client is assuming an array, the application will likely break. Therefore, any consumer or client would need to be aware of the special case where the vector becomes a primitive, and explicitly take this exception into account when processing the result. When the client fails to do so and proceeds as usual, it will probably call an iterator or loop method on a primitive value, resulting in the obvious errors. To avoid this, jsonlite uses consistent encoding schemes which do not depend on variable object properties such as its length. Hence, a vector is always encoded as an array, even when it is of length 0 or 1.

4.2.2 Matrices

Arguably one of the strongest sides of R is its ability to interface libraries for basic linear algebra subprograms (Lawson et al., 1979) such as LAPACK (Anderson et al., 1987). These libraries provide well tuned, high performance implementations of important linear algebra operations to calculate anything from inner products and eigen values to singular value decompositions, which are in turn building blocks of statistical methods such as linear regression or principal component analysis. Linear algebra methods operate on *matrices*, making the matrix one of the most central data classes in R. Conceptually, a matrix consists of a 2 dimensional structure of homogeneous values. It is indexed using 2 numbers (or vectors), representing the rows and columns of the matrix respectively.

```
x <- matrix(1:12, nrow = 3, ncol = 4)
print(x)

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

print(x[2, 4])

[1] 11
```

A matrix is stored in memory as a single atomic vector with an attribute called "dim" defining the dimensions of the matrix. The product of the dimensions is equal to the length of the vector.

```
attributes(volcano)

$dim
[1] 87 61
```

```
length(volcano)

[1] 5307
```

Even though the matrix is stored as a single vector, the way it is printed and indexed makes it conceptually a 2 dimensional structure. In `jsonlite` a matrix maps to an array of equal-length subarrays:

```
x <- matrix(1:12, nrow = 3, ncol = 4)
cat(toJSON(x))

[ [ 1, 4, 7, 10 ], [ 2, 5, 8, 11 ], [ 3, 6, 9, 12 ] ]
```

We expect this representation will be the most intuitive to interpret, also within languages that do not have a native notion of a matrix. Note that even though `R` stores matrices in *column major* order, `jsonlite` encodes matrices in *row major* order. This is a more conventional and intuitive way to represent matrices and is consistent with the row-based encoding of data frames discussed in the next section. When the `JSON` string is properly indented (recall that white space and line breaks are optional in `JSON`), it looks very similar to the way `R` prints matrices:

```
[ [ 1, 4, 7, 10 ],
  [ 2, 5, 8, 11 ],
  [ 3, 6, 9, 12 ] ]
```

Because the matrix is implemented in `R` as an atomic vector, it automatically inherits the conventions mentioned earlier with respect to edge cases and missing values:

```
x <- matrix(c(1, 2, 4, NA), nrow = 2)
cat(toJSON(x))

[ [ 1, 4 ], [ 2, "NA" ] ]

cat(toJSON(x, na = "null"))

[ [ 1, 4 ], [ 2, null ] ]

cat(toJSON(matrix(pi)))

[ [ 3.14 ] ]
```

4.2.2.1 Matrix row and column names

Besides the `"dim"` attribute, the matrix class has an additional, optional attribute: `"dimnames"`. This attribute holds names for the rows and columns in the matrix. However, we decided not to include this information in the default `JSON` mapping for matrices for several reasons. First of all, because this attribute is optional, either row or column names or both could be `NULL`. This makes it difficult to define a practical mapping that covers all cases with and without row and/or column names. Secondly, the names in matrices are mostly there for annotation only; they are not actually used in calculations. The linear algebra subroutines mentioned before completely ignore them, and never include any names in their output. So there is often little purpose of setting names in the first place, other than annotation.

When row or column names of a matrix seem to contain vital information, we might want to transform the data into a more appropriate structure. Wickham (2014) calls this *"tidying"* the data and outlines best practices on storing statistical data in its most appropriate form. He lists the issue where *"column headers are* *values, not variable names"* as the most common source of untidy data. This often happens when the structure is optimized for presentation (e.g. printing), rather than computation. In the following example taken from Wickham, the predictor variable (treatment) is stored in the column headers rather than the actual data. As a result, these values do not get included in the `JSON` output:

```
x <- matrix(c(NA, 1, 2, 5, NA, 3), nrow = 3)
row.names(x) <- c("Joe", "Jane", "Mary")
colnames(x) <- c("Treatment A", "Treatment B")
print(x)

     Treatment A Treatment B
Joe           NA           5
Jane           1          NA
Mary           2           3

cat(toJSON(x))

[ [ "NA", 5 ], [ 1, "NA" ], [ 2, 3 ] ]
```

Wickham recommends that the data be *melted* into its *tidy* form. Once the data is tidy, the `JSON` encoding will naturally contain the treatment values:

```
library(reshape2)
y <- melt(x, varnames = c("Subject", "Treatment"))
print(y)

  Subject   Treatment value
1     Joe Treatment A    NA
2    Jane Treatment A     1
3    Mary Treatment A     2
4     Joe Treatment B     5
5    Jane Treatment B    NA
6    Mary Treatment B     3
```

```
cat(toJSON(y, pretty = TRUE))

[
  {
    "Subject" : "Joe",
    "Treatment" : "Treatment A"
  },
  {
    "Subject" : "Jane",
    "Treatment" : "Treatment A",
    "value" : 1
  },
  {
    "Subject" : "Mary",
    "Treatment" : "Treatment A",
    "value" : 2
  },
  {
    "Subject" : "Joe",
    "Treatment" : "Treatment B",
    "value" : 5
  },
  {
    "Subject" : "Jane",
    "Treatment" : "Treatment B"
  },
  {
    "Subject" : "Mary",
    "Treatment" : "Treatment B",
    "value" : 3
  }
]
```

In some other cases, the column headers actually do contain variable names, and melting is inappropriate. For data sets with records consisting of a set of named columns (fields), R has more natural and flexible class: the data-frame. The toJSON method for data frames (described later) is more suitable when we want to refer to rows or fields by their name. Any matrix can easily be converted to a data-frame using the as.data.frame function:

```
cat(toJSON(as.data.frame(x), pretty = TRUE))

[
  {
    "$row" : "Joe",
    "Treatment B" : 5
  },
  {
    "$row" : "Jane",
    "Treatment A" : 1
  },
  {
    "$row" : "Mary",
    "Treatment A" : 2,
    "Treatment B" : 3
  }
]
```

For some cases this results in the desired output, but in this example melting seems more appropriate.

4.2.3 Lists

The list is the most general purpose data structure in R. It holds an ordered set of elements, including other lists, each of arbitrary type and size. Two types of lists are distinguished: named lists and unnamed lists. A list is considered a named list if it has an attribute called "names". In practice, a named list is any list for which we can access an element by its name, whereas elements of an unnamed lists can only be accessed using their index number:

```r
mylist1 <- list(foo = 123, bar = 456)
print(mylist1$bar)

[1] 456

mylist2 <- list(123, 456)
print(mylist2[[2]])

[1] 456
```

4.2.3.1 Unnamed lists

Just like vectors, an unnamed list maps to a `JSON` array:

```r
cat(toJSON(list(c(1, 2), "test", TRUE, list(c(1, 2)))))

[ [ 1, 2 ], [ "test" ], [ true ], [ [ 1, 2 ] ] ]
```

Note that even though both vectors and lists are encoded using `JSON` arrays, they can be distinguished from their contents: an `R` vector results in a `JSON` array containing only primitives, whereas a list results in a `JSON` array containing only objects and arrays. This allows the `JSON` parser to reconstruct the original type from encoded vectors and arrays:

```r
x <- list(c(1, 2, NA), "test", FALSE, list(foo = "bar"))
identical(fromJSON(toJSON(x)), x)

[1] TRUE
```

The only exception is the empty list and empty vector, which are both encoded as `[ ]` and therefore indistinguishable, but this is rarely a problem in practice.

4.2.3.2 Named lists

A named list in `R` maps to a `JSON` *object*:

```
cat(toJSON(list(foo = c(1, 2), bar = "test")))

{ "foo" : [ 1, 2 ], "bar" : [ "test" ] }
```

Because a list can contain other lists, this works recursively:

```
cat(toJSON(list(foo=list(bar=list(baz=pi)))))

{ "foo" : { "bar" : { "baz" : [ 3.14 ] } } }
```

Named lists map almost perfectly to `JSON` objects with one exception: list elements can have empty names:

```
x <- list(foo = 123, "test", TRUE)
attr(x, "names")

[1] "foo" ""    ""

x$foo

[1] 123

x[[2]]

[1] "test"
```

In a `JSON` object, each element in an object must have a valid name. To ensure this property, `jsonlite` uses the same solution as the `print` method, which is to fall back on indices for elements that do not have a proper name:

```
x <- list(foo = 123, "test", TRUE)
print(x)

$foo
[1] 123


[[2]]
[1] "test"


[[3]]
[1] TRUE

cat(toJSON(x))

{ "foo" : [ 123 ], "2" : [ "test" ], "3" : [ true ] }
```

This behavior ensures that all generated JSON is valid, however named lists with empty names should be avoided where possible. When actually designing R objects that should be interoperable, it is recommended that each list element is given a proper name.

4.2.4 Data frame

The data frame is perhaps the most central data structure in R from the user point of view. This class holds tabular data in which each column is named and (usually) homogeneous. Conceptually it is very similar to a table in relational data bases such as MySQL, where *fields* are referred to as *column names*, and *records* are called *rows*. Like a matrix, a data frame can be subsetted with two indices, to extract certain rows and columns of the data:

```
is(iris)

[1] "data.frame" "list"       "oldClass"   "vector"
```

```
names(iris)

[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
[5] "Species"

print(iris[1:3, c(1, 5)])

  Sepal.Length Species
1          5.1  setosa
2          4.9  setosa
3          4.7  setosa

print(iris[1:3, c("Sepal.Width", "Species")])

  Sepal.Width Species
1         3.5  setosa
2         3.0  setosa
3         3.2  setosa
```

For the previously discussed classes such as vectors and matrices, behavior of `jsonlite` was quite similar to the other available packages that implement `toJSON` and `toJSON` functions, with only minor differences for missing values and edge cases. But when it comes to data frames, `jsonlite` takes a completely different approach. The behavior of `jsonlite` is designed for compatibility with conventional ways of encoding table-like structures outside the `R` community. The implementation is more involved, but results in a powerful and more natural way of representing data frames in `JSON`.

4.2.4.1 Column based versus row based tables

Generally speaking, tabular data structures can be implemented in two different ways: in a column based, or row based fashion. A column based structure consists of a named collection of equal-length, homogeneous arrays representing the table columns. In a row-based structure on the other hand, the table is implemented as a set of heterogeneous associative arrays representing table rows with field values for each particular record. Even though most languages provide flexible and abstracted interfaces that hide these implementation details from the user, they can have huge implications for performance. A column based structure is efficient for inserting or extracting certain columns of the data, but it is inefficient for manipulating individual rows. For example to insert a single row somewhere in the middle, each of the columns has to be sliced and stitched back together. For row-based implementations, it is the exact other way around: we can easily manipulate a particular record, but to insert/extract a whole column we would need to iterate over all records in the table and read/modify the appropriate field in each of them.

The data frame class in `R` is implemented in a column based fashion: it constitutes of a `named list` of equal-length vectors. Thereby the columns in the data frame naturally inherit the properties from atomic vectors discussed before, such as homogeneity, missing values, etc. Another argument for column-based implementation is that statistical methods generally operate on columns. For example, the `lm` function fits a *linear regression* by extracting the columns from a data frame as specified by the `formula` argument. `R` simply binds the specified columns together into a matrix X and calls out to a highly optimized `FORTRAN` subroutine to calculate the OLS estimates $\hat{\beta} = (X^T X)X^T y$ using the QR factorization of X. Many other statistical modeling functions follow similar steps, and are computationally efficient because of the column-based data storage in `R`.

Unfortunately `R` is an exception in its preference for column-based storage: most languages, systems, databases, `API`'s, etc, are optimized for record based operations. For this reason, the conventional way to store and communicate tabular data in `JSON` seems to almost exclusively row based. This discrepancy presents various complications when converting between data frames and JSON. The remaining of this section discusses details and challenges of consistently mapping record based JSON data as frequently encountered on the web, into column-based data frames which are convenient for statistical computing.

4.2.4.2 Row based data frame encoding

The encoding of data frames is one of the major differences between jsonlite and implementations from other currently available packages. Instead of using the column-based encoding also used for lists, jsonlite maps data frames by default to an array of records:

```
cat(toJSON(iris[1:2, ], pretty = TRUE))
```

```
[
  {
    "Sepal.Length" : 5.1,
    "Sepal.Width" : 3.5,
    "Petal.Length" : 1.4,
    "Petal.Width" : 0.2,
    "Species" : "setosa"
  },
  {
    "Sepal.Length" : 4.9,
    "Sepal.Width" : 3,
    "Petal.Length" : 1.4,
    "Petal.Width" : 0.2,
    "Species" : "setosa"
  }
]
```

This output looks a bit like a list of named lists. However, there is one major difference: the individual records contain JSON primitives, whereas lists always contain JSON objects or arrays:

```
cat(toJSON(list(list(Species = "Foo", Width = 21)), pretty = TRUE))

[
  {
    "Species" : [
      "Foo"
    ],
    "Width" : [
      21
    ]
  }
]
```

This leads to the following convention: when encoding R objects, JSON primitives only appear in vectors and data-frame rows. Primitives within a JSON array indicate a vector, and primitives appearing inside a JSON object indicate a data-frame row. A JSON encoded `list`, (named or unnamed) will never contain JSON primitives. This is a subtle but important convention that helps to distinguish between R classes from their JSON representation, without explicitly encoding any metadata.

4.2.4.3 Missing values in data frames

The section on atomic vectors discussed two methods of encoding missing data appearing in a vector: either using strings or using the JSON `null` type. When a missing value appears in a data frame, there is a third option: simply not include this field in JSON record:

```
x <- data.frame(foo = c(FALSE, TRUE, NA, NA), bar = c("Aladdin", NA, NA, "Mario"))
print(x)

    foo     bar
```

```
1 FALSE Aladdin
2  TRUE    <NA>
3    NA    <NA>
4    NA   Mario

cat(toJSON(x, pretty = TRUE))

[
  {
    "foo" : false,
    "bar" : "Aladdin"
  },
  {
    "foo" : true
  },
  {},
  {
    "bar" : "Mario"
  }
]
```

The default behavior of `jsonlite` is to omit missing data from records in a data frame. This seems to be the most conventional method used on the web, and we expect this encoding will most likely lead to the correct interpretation of *missingness*, even in languages without an explicit notion of `NA`.

4.2.4.4 Relational data: nested records

Nested datasets are somewhat unusual in `R`, but frequently encountered in `JSON`. Such structures do not really fit the vector based paradigm which makes them harder to manipulate in `R`. However, nested structures are too common in `JSON` to ignore, and with a little work most cases still map to a data frame quite nicely. The most common scenario is a dataset in which a certain field within each record contains a *subrecord* with additional fields. The `jsonlite` implementation maps these subrecords to a nested data frame. Whereas the data frame class usually consists of vectors, technically a column can also be list or another data frame with matching dimension (this stretches the meaning of the word “column” a bit):

```r
options(stringsAsFactors=FALSE)
x <- data.frame(driver = c("Bowser", "Peach"),
  occupation = c("Koopa", "Princess"))
x$vehicle <- data.frame(model = c("Piranha Prowler", "Royal Racer"))
x$vehicle$stats <- data.frame(speed = c(55, 34), weight = c(67, 24),
  drift = c(35, 32))
str(x)
```

```
'data.frame': 2 obs. of  3 variables:
 $ driver    : chr  "Bowser" "Peach"
 $ occupation: chr  "Koopa" "Princess"
 $ vehicle   :'data.frame': 2 obs. of  2 variables:
  ..$ model: chr  "Piranha Prowler" "Royal Racer"
  ..$ stats:'data.frame': 2 obs. of  3 variables:
  .. ..$ speed : num  55 34
  .. ..$ weight: num  67 24
  .. ..$ drift : num  35 32
```

```r
cat(toJSON(x, pretty=TRUE))
```

```
[
  {
    "driver" : "Bowser",
    "occupation" : "Koopa",
    "vehicle" : {
      "model" : "Piranha Prowler",
      "stats" : {
        "speed" : 55,
        "weight" : 67,
        "drift" : 35
      }
    }
  },
  {
    "driver" : "Peach",
    "occupation" : "Princess",
    "vehicle" : {
      "model" : "Royal Racer",
      "stats" : {
        "speed" : 34,
        "weight" : 24,
        "drift" : 32
      }
    }
  }
]

myjson <- toJSON(x)
y <- fromJSON(myjson)
identical(x,y)

[1] TRUE
```

When encountering `JSON` data containing nested records on the web, chances are that these data were generated from *relational* database. The `JSON` field containing a subrecord represents a *foreign key* pointing to a record in an external table. For the purpose of encoding these into a single `JSON` structure, the tables were joined into a nested structure. The directly nested subrecord represents a *one-to-one* or *many-to-one* relation between the parent and child table, and is most naturally stored in `R` using a nested data frame. In the example above, the `vehicle` field points to a table of vehicles, which in turn contains a `stats` field pointing to a table of stats. When there is no more than one subrecord for each record, we easily *flatten* the structure into a single non-nested data frame.

```
y <- fromJSON(myjson, flatten = TRUE)
str(y)

'data.frame': 2 obs. of  6 variables:
 $ driver              : chr  "Bowser" "Peach"
 $ occupation          : chr  "Koopa" "Princess"
 $ vehicle.model       : chr  "Piranha Prowler" "Royal Racer"
 $ vehicle.stats.speed : num  55 34
 $ vehicle.stats.weight: num  67 24
 $ vehicle.stats.drift : num  35 32
```

4.2.4.5 Relational data: nested tables

The one-to-one relation discussed above is relatively easy to store in `R`, because each record contains at most one subrecord. Therefore we can use either a nested data frame, or flatten the data frame. However, things get more difficult when `JSON` records contain a field with a nested array. Such a structure appears in relational data in case of a *one-to-many* relation. A standard textbook illustration is the relation between authors and titles. For example, a field can contain an array of values:

```
x <- data.frame(author = c("Homer", "Virgil", "Jeroen"))
x$poems <- list(c("Iliad", "Odyssey"), c("Eclogues", "Georgics", "Aeneid"),
  vector());
names(x)

[1] "author" "poems"

cat(toJSON(x, pretty = TRUE))

[
  {
    "author" : "Homer",
    "poems" : [
      "Iliad",
      "Odyssey"
    ]
  },
  {
    "author" : "Virgil",
    "poems" : [
      "Eclogues",
      "Georgics",
      "Aeneid"
    ]
  },
  {
    "author" : "Jeroen",
    "poems" : []
  }
]
```

As can be seen from the example, the way to store this in a data frame is using a list of character vectors. This works, and although unconventional, we can still create and read such structures in R relatively easily. However, in practice the one-to-many relation is often more complex. It results in fields containing a *set of records*. In R, the only way to model this is as a column containing a list of data frames, one separate data frame for each row:

```r
x <- data.frame(author = c("Homer", "Virgil", "Jeroen"))
x$poems <- list(
  data.frame(title=c("Iliad", "Odyssey"), year=c(-1194, -800)),
  data.frame(title=c("Eclogues", "Georgics", "Aeneid"), year=c(-44, -29, -19)),
  data.frame()
)
cat(toJSON(x, pretty=TRUE))
```

```
[
  {
    "author" : "Homer",
    "poems" : [
      {
        "title" : "Iliad",
        "year" : -1194
      },
      {
        "title" : "Odyssey",
        "year" : -800
      }
    ]
  },
  {
    "author" : "Virgil",
    "poems" : [
      {
        "title" : "Eclogues",
        "year" : -44
      },
      {
        "title" : "Georgics",
        "year" : -29
      },
      {
        "title" : "Aeneid",
        "year" : -19
      }
    ]
  },
  {
    "author" : "Jeroen",
    "poems" : []
  }
]
```

Because R doesn't have native support for relational data, there is no natural class to store such structures. The best we can do is a column containing a list of sub-dataframes. This does the job, and allows the R user to access or generate nested JSON structures. However, a data frame like this cannot be flattened, and the class does not guarantee that each of the individual nested data frames contain the same fields, as would be the case in an actual relational data base.

4.3 Structural consistency and type safety in dynamic data

Systems that automatically exchange information over some interface, protocol or API require well defined and unambiguous meaning and arrangement of data. In order to process and interpret input and output, contents must obey a steady structure. Such structures are usually described either informally in documentation or more formally in a schema language. The previous section emphasized the importance of consistency in the mapping between JSON data and R classes. This section takes a higher level view and explains the importance of structure consistency for dynamic data. This topic can be a bit subtle because it refers to consistency among different instantiations of a JSON structure, rather than a single case. We try to clarify by breaking down the concept into two important parts, and illustrate with analogies and examples from R.

4.3.1 Classes, types and data

Most object-oriented languages are designed with the idea that all objects of a certain class implement the same fields and methods. In strong-typed languages such as `S4` or `Java`, names and types of the fields are formally declared in a class definition. In other languages such as `S3` or `JavaScript`, the fields are not enforced by the language but rather at the discretion of the programmer. One way or another they assume that members of a certain class agree on field names and types, so that the same methods can be applied to any object of a particular class. This basic principle holds for dynamic data exactly the same way as for objects. Software that process dynamic data can only work reliably if the various elements of the data have consistent names and structure. Consensus must exist between the different parties on data that is exchanged as part an interface or protocol. This requires the structure to follow some sort of template that specifies which attributes can appear in the data, what they mean and how they are composed. Thereby each possible scenario can be accounted for in the software so that data can be interpreted and processed appropriately with no exceptions during runtime.

Some data interchange formats such as `XML` or `Protocol Buffers` take a formal approach to this matter, and have well established *schema languages* and *interface description languages*. Using such a meta language it is possible to define the exact structure, properties and actions of data interchange in a formal arrangement. However, in `JSON`, such formal definitions are relatively uncommon. Some initiatives for `JSON` schema languages exist (Galiegue and Zyp, 2013), but they are not very well established and rarely seen in practice. One reason for this might be that defining and implementing formal schemas is complicated and a lot of work which defeats the purpose of using an lightweight format such as `JSON` in the first place. But another reason is that it is often simply not necessary to be overly formal. The `JSON` format is simple and intuitive, and under some general conventions, a well chosen example can suffice to characterize the structure. This section describes two important rules that are required to ensure that data exchange using JSON is type safe.

4.3.2 Rule 1: Fixed keys

When using JSON without a schema, there are no restrictions on the keys (field names) that can appear in a particular object. However, a source of data that returns a different set of keys every time it is called makes it very difficult to write software to process these data. Hence, the first rule is to limit JSON interfaces to a finite set of keys that are known *a priory* by all parties. It can be helpful to think about this in analogy with for example a relational database. Here, the database model separates the data from metadata. At run time, records can be inserted or deleted, and a certain query might return different content each time it is executed. But for a given query, each execution will return exactly the same *field names*; hence as long as the table definitions are unchanged, the *structure* of the output consistent. Client software needs this structure to validate input, optimize implementation, and process each part of the data appropriately. In JSON, data and metadata are not formally separated as in a database, but similar principles that hold for fields in a database, apply to keys in dynamic JSON data.

A beautiful example of this in practice was given by Mike Dewar at the New York Open Statistical Programming Meetup on Jan. 12, 2012 (Dewar, 2012). In his talk he emphasizes to use JSON keys only for *names*, and not for *data*. He refers to this principle as the "golden rule", and explains how he learned his lesson the hard way. In one of his early applications, timeseries data was encoded by using the epoch timestamp as the JSON key. Therefore the keys are different each time the query is executed:

```
[
  { "1325344443" : 124 },
  { "1325344456" : 131 },
  { "1325344478" : 137 }
]
```

Even though being valid JSON, dynamic keys as in the example above are likely to introduce trouble. Most software will have great difficulty processing these values if we can not specify the keys in the code. Moreover when documenting the API, either informally or formally using a schema language, we need to describe for each property in the data what the value means and is composed of. Thereby a client or consumer can implement code that interprets and process each element in the data in an appropriate manner. Both the documentation and interpretation of JSON data rely on fixed keys with well defined meaning. Also note that the structure is difficult to extend in the future. If we want to add an additional property to each observation, the entire structure needs to change. In his talk, Dewar explains that life gets much easier when we switch to the following encoding:

```
[
  { "time": "1325344443" : "price": 124 },
  { "time": "1325344456" : "price": 131 },
  { "time": "1325344478" : "price": 137 }
]
```

This structure will play much nicer with existing software that assumes fixed keys. Moreover, the structure can easily be described in documentation, or captured in a schema. Even when we have no intention of writing documentation or a schema for a dynamic JSON source, it is still wise to design the structure in such away that it *could* be described by a schema. When the keys are fixed, a well chosen example can provide all the information required for the consumer to implement client code. Also note that the new structure is extensible: additional properties can be added to each observation without breaking backward compatibility.

In the context of R, consistency of keys is closely related to Wikcham's concept of *tidy data* discussed earlier. Wickham states that the most common reason for messy data are column headers containing values instead of variable names. Column headers in tabular datasets become keys when converted to JSON. Therefore, when headers are actually values, JSON keys contain in fact data and can become unpredictable. The cure to inconsistent keys is almost always to tidy the data according to recommendations given by Wickham (2014).

4.3.3 Rule 2: Consistent types

In a strong typed language, fields declare their class before any values are assigned. Thereby the type of a given field is identical in all objects of a particular class, and arrays only contain objects of a single type. The S3 system in R is weakly typed and puts no formal restrictions on the class of a certain properties, or the types of objects that can be combined into a collection. For example, the list below contains a character vector, a numeric vector and a list:

```r
# Heterogeneous lists are bad!
x <- list("FOO", 1:3, list(bar = pi))
cat(toJSON(x))

[ [ "FOO" ], [ 1, 2, 3 ], { "bar" : [ 3.14 ] } ]
```

However even though it is possible to generate such JSON, it is bad practice. Fields or collections with ambiguous object types are difficult to describe, interpret and process in the context of inter-system communication. When using JSON to exchange dynamic data, it is important that each property and array is *type consistent*. In dynamically typed languages, the programmer needs to make sure that properties are of the correct type before encoding into JSON. For R, this means that the unnamed lists type is best avoided when designing interoperable structures because this type is not homogeneous.

Note that consistency is somewhat subjective as it refers to the *meaning* of the elements; they do not necessarily have precisely the same structure. What is important is to keep in mind that the consumer of the data can interpret and process each element identically, e.g. iterate over the elements in the collection and apply the same method to each of them. To illustrate this, lets take the example of the data frame:

```r
# conceptually homogenous array
x <- data.frame(name = c("Jay", "Mary", NA, NA), gender = c("M", NA, NA, "F"))
cat(toJSON(x, pretty = TRUE))
```

```
[
  {
    "name" : "Jay",
    "gender" : "M"
  },
  {
    "name" : "Mary"
  },
  {},
  {
    "gender" : "F"
  }
]
```

The JSON array above has 4 elements, each of which a JSON object. However, due to the NA values, some records have more fields than others. But as long as they are conceptually the same type (e.g. a person), the consumer can iterate over the elements to process each person in the set according to a predefined action. For example each element could be used to construct a `Person` object. A collection of different object classes should be separated and organized using a named list:

```
x <- list(
  humans = data.frame(name = c("Jay", "Mary"), married = c(TRUE, FALSE)),
  horses = data.frame(name = c("Star", "Dakota"), price = c(5000, 30000))
)
cat(toJSON(x, pretty=TRUE))

{
  "humans" : [
    {
      "name" : "Jay",
      "married" : true
    },
    {
      "name" : "Mary",
      "married" : false
    }
  ],
  "horses" : [
    {
      "name" : "Star",
      "price" : 5000
    },
    {
      "name" : "Dakota",
      "price" : 30000
    }
  ]
}
```

This might seem obvious, but dynamic languages such as `R` can make it dangerously tempting to generate data containing mixed-type collections. Such inconsistent typing makes it very difficult to consume the data and creates a likely source of nasty bugs. Using consistent field names/types and homogeneous `JSON` arrays is a strong convention among public `JSON API`'s, for good reasons. We recommend `R` users to respect these conventions when generating `JSON` data in `R`.

CHAPTER 5

Possible Directions for Improving Dependency Versioning in R

5.1 Package management in R

One of the most powerful features of R is its infrastructure for contributed code (Fox, 2009). The base R software suite that is released several times per year ships with the *base* and *recommended* packages and provides a solid foundation for statistical computing. However, most R users will quickly resort to the package manager and install packages contributed by other users. By default, these packages are installed from the "Comprehensive R Archive Network" (`CRAN`), featuring over 4300 contributed packages as of 2013. In addition, other repositories like BioConductor (Gentleman et al., 2004) and Github (Dabbish et al., 2012) are hosting a respectable number of packages as well.

The *R Core team* has done a tremendous job in coordinating the development of the base software along with providing, supporting, and maintaining an infrastructure for contributed code. The system for sharing and installing contributed packages is easily taken for granted, but could in fact not survive without the commitment and daily efforts from the repository maintainers. The process from submission to publication of a package involves several manual steps needed to ensure that all published packages meet standards and work as expected, on a variety of platforms, architectures and R versions. In spite of rapid growth and limited resources, CRAN has managed to maintain high standards on the quality of packages. Before continuing, we want to express appreciation for the countless hours invested by volunteers in organizing this unique forum for statistical software. They facilitate the innovation and collaboration in our field, and unite the community in creating software that is both of the highest quality and publicly available. We want to emphasize that suggestions made in this paper are in no way intended as criticism on the status quo. If anything, we hope that our ideas help address some challenges to support further growth without having to compromise on the open and dynamic nature of the infrastructure.

5.1.1 The dependency network

Most R packages depend on one or more other packages, resulting in a complex network of recursive dependencies. Each package includes a `DESCRIPTION` file which allows for declaration of several types of dependencies, including `Depends`, `Imports`, `Suggests` and `Enhances`. Based on the type of dependency relationship, other packages are automatically installed, loaded and/or attached with the requested package. Package management is also related to the issue of *namespacing*, because different packages can use identical names for objects. The `NAMESPACE` file allows the developer to explicitly define objects to be exported or imported from other packages. This prevents the need to attach all dependencies and lookup variables at runtime, and thereby decreases chances of masking and naming-conflicts. Unfortunately, many packages are not taking advantage of this feature, and thereby force R to attach all dependencies, unnecessarily filling the search path of a session with packages that the user hasn't asked for. However, this is not the primary focus of this paper.

5.1.2 Package versioning

Even though CRAN consistently archives older versions of every package when updates are published, the R software itself takes limited advantage of this archive. The package manager identifies packages by name only when installing or loading a package. The `install.packages` function downloads and installs the *current* version of a CRAN package into a single global library. This library contains a single version of each package. If a previous version of the package is already installed on the system, it is overwritten without warning. Similarly, the `library` function will load the earliest found package with a matching name.

The `DESCRIPTION` file does allow the package author to specify a certain version of a dependency by postfixing the package name with `>=`, `<=` or `==` and a version string. However, using this feature is actually dangerous because R might not be able to satisfy these conditions, causing errors. This is again the result of R libraries, sessions and repositories being limited to a single current version of each package. When a package would require a version of a dependency that is not already installed or current on CRAN, it can not be resolved automatically. Furthermore, upgrading a package in the global library to the current CRAN version might break other packages that require the previously installed version. Experienced R users might try to avoid such problems by manually maintaining separate libraries for different tasks and projects. However, R can still not have multiple versions of a package loaded concurrently. This is perhaps the most fundamental problem because it is nearly impossible to work around. If package authors would actually declare specific versions of dependencies, any two packages requiring different versions of one and the same dependency will conflict and cannot be used together. In practice, this limitation discourages package authors to be explicit about dependency versions. The `>=` operator is used by some packages, but it only checks if an installed dependency is outdated and needs to be synchronized with CRAN. It still assumes that any current of future version will suffice, and does not protect packages from breaking when their dependency packages change. The `<=` and `==` operators are barely used at all.

When identifying a package by its name only, we implicitly make the assumption that different versions of the package are interchangeable. This basic assumption has far-reaching implications and consequences on the distributed development process and reliability of the software as a whole. In the context of the increasingly large pool of inter-dependent packages, violations of this assumption are becoming increasingly apparent and problematic. In this paper we explore this problem is greater detail, and try to make a case for moving away from this assumption, towards systematic versioning of dependency relationships. The term *dependency* in this context does not exclusively refer to formally defined relations between R packages. Our interpretation is a bit more general in the sense that any R script, Sweave document, or third party application *depends* on R and certain packages that are needed to make it function. The paper is largely motivated by personal experiences, as we have come to believe that limitations of the current dependency system are underlying multiple problems that R users and developers might experience. Properly addressing these concerns could resolve several lingering issues at once, and make R a more reliable and widely applicable analytical engine.

5.2 Use cases

A dependency defines a relationship wherein a certain piece of software requires some other software to run or compile. However, software constantly evolves, and in the open source world this happens largely unmanaged. Consequently, any software library might actually be something different today than it was yesterday. Hence, solely defining the dependency relationship in terms of the name of the software is often insufficient. We need to be more specific, and declare explicitly which version(s), branch(es) or release(s) of the other software package will make our program work. This is what we will refer to as *depencency versioning*.

This problem is not at all unique to R; in fact a large share of this paper consist of taking a closer look at how other open source communities are managing this process, and if some of their solutions could apply to R as well. But first we will elaborate a bit further on how this problem exactly appears in the context of R. This section describes three use cases that reveal some limitations of the current system. These use cases delineate the problem and lead towards suggestions for improvements in subsequent sections.

5.2.1 Case 1: Archive / repository maintenance

A medium to large sized repository with thousands of packages has a complicated network of dependencies between packages. CRAN is designed to consider the very latest version of every package as the only *current* version. This design relies on the assumption that at any given time, the latest versions of all packages are compatible. Therefore, R's built-in package manager can simply download and install the current versions of all dependencies along with the requested package, which seems convenient. However, to developers this means that every package *update* needs to maintain full backward compatibility with all previous versions. No version can introduce any breaking changes, because other packages in the repository might be relying on things in a certain way. Functions or objects may never be removed or modified; names, arguments, behavior, etc, must remain the same. As the dependency network gets larger and more complex, this policy becomes increasingly vulnerable. It puts a heavy burden on contributing developers, especially the popular ones, and results in increasingly large packages that are never allowed to deprecate or clean up old code and functionality.

In practice, the assumption is easily violated. Every time a package update is pushed to CRAN, there is a real chance of some reverse dependencies failing due to a breaking change. In the case of the most popular packages, the probability of this happening is often closer to 1 than to 0, regardless of the author. Uwe Ligges has stated in his keynote presentation at useR that CRAN automatically detects some of these problems by rebuilding every package up in the dependency tree. However, only a small fraction of potential problems reveal themselves during the build of a package, and when found, there is no obvious solution. One recent example was the forced roll-back of the `ggplot2` (Wickham, 2009) update to version 0.9.0, because the introduced changes caused several other packages to break. The author of the `ggplot2` package has since been required to announce upcoming updates to authors of packages that depend on `ggplot2`, and provide a release candidate to test compatibility. The dependent packages are then required to synchronize their releases if any problems arise. However, such manual solutions are far from flawless and put even more work on the shoulders of contributing developers. It is doubtful that all package authors on CRAN have time and resources to engage in an extensive dialogue with other maintainers for each update of a package. We feel strongly that a more systematic solution is needed to guarantee that software published on CRAN keeps working over time; current as well as older versions.

When the repository reaches a critical size, and some packages collect hundreds of reverse dependencies, we have little choice but to acknowledge the fact that every package has only been developed for, and tested with certain versions of its dependencies. A policy of assuming that any current or future version of a dependency should suffice is dangerous and sets the wrong incentives for package authors. It discourages change, refactoring or cleanup, and results in packages accumulating an increasingly heavy body of legacy code. And as the repository grows, it is inevitable that packages will nevertheless eventually break as part of the process. What is needed is a redesign that supports the continuous decentralized change of software and helps facilitate more reliable package development.

This is not impossible: there are numerous open source communities managing repositories with more complex dependency structures than CRAN. Although specifics vary, they form interesting role models to our community. As we will see later on, a properly archived repository can actually come to be a great asset rather than a liability to the developer.

5.2.2 Case 2: Reproducibility

Replication is the ultimate standard by which scientific claims are judged. However, complexity of data and methods can make this difficult to achieve computational science (Peng, 2011). As a leader in scientific computing, R takes a pioneering role in providing a system that encourages researchers to strive towards the gold standard. The CRAN Task View on Reproducible Research states that:

> *The goal of reproducible research is to tie specific instructions to data analysis and experimental data so that scholarship can be recreated, better understood and verified.*

In R, reproducible research is largely facilitated using literate programming techniques implemented in packages like `Sweave` that mix (weave) R code with LaTeX-markup to create a "reproducible document" (Leisch, 2002). However, those ever faced with the task of actually reproducing such a document might have experienced that the Sweave file does not always compile out of the box. Especially if it was written several years ago and loads some contributed packages, chances are that essential things have changed in the software since the document was created. When we find ourselves in such a situation, recovering the packages needed to reproduce the document might turn out to be non-trivial.

An example: suppose we would like to reproduce a Sweave document which was created with R 2.13 and loads the `caret` package (Kuhn, 2013). If no further instructions are provided, this means that any of the approximately 25 releases of `caret` in the life cycle of R 2.13 (April 2011 to February 2012) could have been used, making reproducibility unlikely. Sometimes authors add comments in the code where the package is loaded, stating that e.g. `caret 4.78` was used. However, this information might also turn out to be insufficient: `caret` depends on 4 packages, and suggests another 59 packages, almost all of which have had numerous releases in R 2.13 time frame. Consequently, `caret 4.78` might not work anymore because of changes in these dependencies. We then need to do further investigation to figure out which versions of the dependency packages were current at the time of the `caret 4.78` release. Instead, lets assume that the prescient researcher anticipated all of this, and saved the full output of `sessionInfo()` along with the Sweave document, directly after it was compiled. This output lists the version of each loaded package in the active R session. We could then proceed by manually downloading and installing R 2.13 along with all of the required packages from the archive. However, users on a commercial operating systems might be up for another surprise: unlike source packages, binary packages are not fully archived. For example, the only binary builds available for R 2.13 are respectively `caret 5.13` on Windows, and `caret 5.14` on OSX. Most likely, they will face the task of rebuilding each of the required packages from source in an attempt to reconstruct the environment of the author.

Needless to say, this situation is suboptimal. For manually compiling a single Sweave document we might be willing to make this effort, but it does not provide a solid foundation for systematic or automated reproducible software practices. To make results generated by R more reproducible, we need better conventions and/or native support that is both explicit and specific about contributed code. For an R script or Sweave document to stand the test of time, it should work at least on the same version of R that was used by the author. In this respect, R has higher requirements on versioning than other software. Reproducible research does not just require a version that will make things work, but one that generates exactly the same output. In order to systematically reproduce results R, package versions either need to be standardized, or become a natural part of the language. We realize this will not archive perfect reproducibility, as problems can still arise due to OS or compiler specific behavior. However, it will be a major step forward that has the potential of turning reproducibility into a natural feature of the software, rather than a tedious exercise.

5.2.3 Case 3: Production applications

R is no longer exclusively used by the local statistician through an interactive console. It is increasingly powering systems, stacks and applications with embedded analytics and graphics. When R is part of say, an application used in hospitals to create on-demand graphics from patient data, the underlying code needs to be stable, reliable, and redistributable. Within such an application, even a minor change in code or behavior can result in complete failure of the system and cannot easily be fixed or debugged. Therefore, when an application is put in production, software has to be completely frozen.

An application that builds on R has been developed and tested with certain versions of the base software and R packages used by the application. In order to put this application in production, exactly these versions need to be shipped, installed and loaded by the application on production servers. Managing, distributing and deploying production software with R is remarkably hard, due to limited native dependency versioning and the single global library design. Administrators might discover that an application that was working in one place doesn't work elsewhere, even though exactly the same operating system, version of R, and installation scripts were used. The problem of course is that the contributed packages constantly change. Problems become more complicated when a machine is hosting many applications that were developed by different people and depend on various packages and package versions.

The default behavior of loading packages from a global library with bleeding edge versions is unsuitable for building applications. Because the CRAN repository has no notion of stable branches, one manually needs to download and install the correct versions of packages in a separate library for each application to avoid conflicts. This is quite tricky and hard to scale when hosting many applications. In practice, application developers might not even be aware of these pitfalls, and design their applications to rely on the default behavior of the package manager. They then find out the hard way that applications start breaking down later on, because of upstream changes or library conflicts with other applications.

5.3 Solution 1: staged distributions

The problem of managing bottom-up decentralized software development is not new; rather it is a typical feature of the open source development process. The remainder of this paper will explore two solutions from other open source communities, and suggest how these might apply to R. The current section describes the more classic solution that relies on staged software *distributions*.

A *software distribution* (also referred to as a *distribution* or a *distro*) is a collection of software components built, assembled and configured so that it can be used essentially "as is" for its intended purpose. Maintainers of distributions do not develop software themselves; they collect software from various sources, package it up and redistribute it as a system. Distributions introduce a formal release cycle on the continuously changing upstream developments and maintainers of a distribution take responsibility for ensuring compatibility of different packages within a certain release of the distribution. Software distributions are most commonly known in the context of free operating systems (BSD, Linux, etc). Staging and shipping software in a distribution has proven to scale well to very large code bases. For example, the popular Debian GNU/Linux distribution (after which

R's package description format was modeled) features over 29000 packages with a large and complex dependency network. No single person is familiar with even a fraction of the code base that is hosted in this repository. Yet through well organized staging and testing, this distribution is known to be one of the most reliable operating systems today, and is the foundation for a large share of the global IT infrastructure.

5.3.1 The release cycle

In a nutshell, a staged distribution release can be organized as follows. At any time, package authors can upload new versions of packages to the *devel* pool, also known as the *unstable* branch. A release cycle starts with distribution maintainers announcing a *code freeze* date, several months in advance. At this point, package authors are notified to ensure that their packages in the unstable branch are up to date, fix bugs and resolve other problems. At the date of the code freeze, a copy (fork) of the unstable repository is made, named and versioned, which goes into the *testing* phase. Software in this branch will then be subject to several iterations of intensive testing and bug fixing, sometimes accompanied by *alpha* or *beta* releases of the distribution. However, software versions in the testing branch will no longer receive any major updates that could potentially have side effects or break other packages. The goal is to converge to increasingly stable set of software. When after several testing rounds the distribution maintainers are confident that all serious problems are fixed, the branch is tagged *stable* and released to the public. Software in a stable release will usually only receive minor non-breaking updates, like important compatibility fixes and security updates. For the next "major release" of any software, the user will have to wait for the next cycle of the distribution. As such, everyone using a certain release of the distribution is using exactly the same versions of all programs and libraries on the system. This is convenient for both users and developers and gives distributions a key role in bringing decentralized open source development efforts together.

5.3.2 R: downstream staging and repackaging

The semi annual releases of the r-base software suite can already be considered as a distribution of the 29 `base` and `recommended` packages. However in the case of R, this collection is limited to software that has been centrally developed and released by the same group of people; it does not include contributed code. Due to the lack of native support for dependency versioning in R, several third party projects have introduced some form of downstream staging in order to create stable, redistributable collections R software. This section lists some examples and explains why this is suboptimal. In the next section we will discuss what would be involved with extending the R release cycle to contributed packages.

One way of staging R packages downstream is by including them in existing software distributions. For example, Eddelbuettel and Blundell (2009) have wrapped some popular CRAN packages into `deb` packages for the Debian and Ubuntu systems. Thereby, pre-compiled binaries are shipped in the distribution along with the R base software, putting version compatibility in the hands of the maintainers (among other benefits). This works well, but requires a lot of effort and commitment from the package maintainer, which is why this has only been done for a small subset of the CRAN packages. Most distributions expect high standards on the quality of the software and package maintenance, which makes this approach hard to scale up to many more packages. Furthermore, we are tied to the release cycle of the distribution, resulting in a somewhat arbitrary and perhaps unfortunate snapshot of CRAN packages when the distribution freezes. Also, different distributions will have different policies on if, when and which packages they wish to ship with their system.

Another approach is illustrated by domain-specific projects like BioConductor (genomic data) and REvolution R Enterprise (big data). Both these systems combine a fixed version of R with a custom library of frozen R packages. In the case of REvolution, the full library is included with the installer; for BioConductor they are provided through a dedicated repository. In both cases, this effectively prevents installed software from being altered unexpectedly by upstream changes. However, this also leads to a split in the community between users of R, BioConductor, and REvolution Enterprise. Because of the differences in libraries, R code is not automatically portable between these systems, leading to fragmentation and duplication of efforts. E.g. BioConductor seems to host many packages that could be more generally useful; yet they are unknown to most users of R. Furthermore, both projects only target a limited set of packages; they still rely on CRAN for the majority of the contributed code.

The goal of staging is to tie a fixed set of contributed packages to a certain release of R. If these decisions are passed down to distributions or organizations, a multitude of local conventions and repositories arises, and different groups of users will still be using different package versions. This leads to unnecessary fragmentation of the community by system, organization, or distribution channel. Moreover, it is often hard to assess compatibility of third party packages, resulting in somewhat arbitrary local decision making. It seems that the people who are in the best position to manage and control compatibility are the package authors themselves. This leads us to conclude that a more appropriate place to organize staging of R packages is further upstream.

5.3.3 Branching and staging in CRAN itself

Given that the community of R contributors evolves mainly around CRAN, the most desirable approach to organizing staging would be by integrating it with the publication process. Currently, CRAN is managed as what distributions would consider a *development* or *unstable* branch. It consists of the pool of *bleeding-edge* versions, straight from package authors. Consequently it is wise to assume that software in this branch might break on a regular basis. Usually, the main purpose of an *unstable* branch is for developers to exchange new versions and test compatibility of software. Regular users obtain software releases from *stable* branches instead. This does not sound unfamiliar: the r-base software also distinguishes between stable versions *r-release* and *r-release-old*, and an unstable development version, *r-devel*.

The fact that R already has an semi-annual release cycle for the 29 `base` and `recommended` packages, would make it relatively straightforward to extend this cycle to CRAN packages. A snapshot of CRAN could be frozen along with every version of *r-release*, and new package updates would only be published to the *r-devel* branch. In practice, this could perhaps quite easily be implemented by creating a directory on CRAN for each release of R, containing symbolic links to the versions of the packages considered *stable* for this release. In the case of binary packages for OSX and Windows, CRAN actually already has separate directories with builds for each release of R. However currently these are not frozen and continuously updated. In a staged repository, newly submitted packages are only build for the current *devel* and *testing* branches; they should not affect *stable* releases. Exceptions to this process could still be granted to authors that need to push an important update or bugfix within a stable branch, commonly referred to as *backporting*, but this should only happen incidentally.

To fully make the transition to a staged CRAN, the default behavior of the package manager must be modified to download packages from the stable branch of the current version of R, rather than the latest development release. As such, all users on a given version of R will be using the same version of each CRAN package, regardless on when it was installed. The user could still be given an option to try and install the development version from the unstable branch, for example by adding an additional parameter to `install.packages` named `devel=TRUE`.

However when installing an unstable package, it must be flagged, and the user must be warned that this version is not properly tested and might not be working as expected. Furthermore, when loading this package a warning could be shown with the version number so that it is also obvious from the output that results were produced using a non-standard version of the contributed package. Finally, users that would always like to use the very latest versions of all packages, e.g. developers, could install the `r-devel` release of R. This version contains the latest commits by R Core and downloads packages from the devel branch on CRAN, but should not be used or in production or reproducible research settings.

5.3.4 Organizational change

Appropriate default behavior of the software is a key element to encourage adoption of conventions and standards in the community. But just as important is communication and coordination between repository maintainers and package authors. To make staging work, package authors must be notified of upcoming deadlines, code freezes or currently broken packages. Everyone must realize that the package version that is current at the time of code freeze, will be used by the majority of users of the upcoming version of R. Updates to already released *stable* branches can only be granted in exceptional circumstances, and must guarantee to maintain full backward compatibility. The policies of the BioConductor project provide a good starting point and could be adapted to work for CRAN.

Transitioning to a system of “stable” and “development” branches in CRAN, where the stable branch is conventional for regular users, could tremendously improve the reliability of the software. The version of the R software itself would automatically imply certain versions of contributed packages. Hence, all that is required to reproduce a Sweave document created several years ago, is which version of R was used to create the document. When deploying an application that depends on R 2.15.2 and various contributed packages, we can be sure that a year later the application can be deployed just as easily, even though the authors of contributed packages used by the application might have decided to implement some breaking changes. And package updates that deprecate old functionality or might break other packages that depend on it, can be uploaded to the *unstable* branch without worries, as the stable branches will remain unchanged and users won't be affected. The authors of the dependent packages that broke due to the update can be warned and will have sufficient time to fix problems before the next *stable* release.

5.4 Solution 2: versioned package management

The previous section described the "classical" solution of creating distributable sets of compatible, stable software. This is a proven approach and has been adopted in some way or another by many open-source communities. However, one drawback of this approach might be that some additional coordination is needed for every release. Another drawback is that it makes the software a bit more conservative, in the sense that regular users will generally be using versions of packages that are at least a couple of months old. The current section describes a different approach to the problem that is used by for example the Javascript community. This method is both reliable and flexible, however would require some more fundamental changes to be implemented in R.

5.4.1 Node.js and NPM

One of the most recent and fastest growing open source communities is that of the node.js software (for short: *node*), a Javascript server system based on the open source engine *V8* from Google. One of the reasons that the community has been able to grow rapidly is because of the excellent package manager and identically named repository, *NPM*. Even though this package manager is only 3 years old, it is currently hosting over 30000 packages with more than a million downloads daily, and has quickly become the standard way of distributing Javascript code. The NPM package manager is a powerful tool for development, publication and deployment of both libraries and applications. NPM addresses some problems that Javascript and R actually have in common, and makes an interesting role model for a modern solution to the problem.

The Javascript community can be described as decentralized, unorganized and highly fragmented development without any quality control authority. Similar to CRAN, NPM basically allows anyone to claim a "package name" and start publishing packages and updates to the repositories. The repository has no notion of branches and simply stores every version of a package indefinitely in its archives. However, a major difference with R is how the package manager handles installation, loading and namespacing of packages.

5.4.2 Dependencies in NPM

Every *NPM* package ships with a file named `package.json`, which is the equivalent of the `DESCRIPTION` in R packages, yet a bit more advanced. An overview of the full feature set of the package manager is beyond the scope of this paper, but the interested reader is highly encouraged to take a look over the fence at this well designed system: `https://npmjs.org/doc/json.html`. The most relevant feature in the context CRAN is how NPM declares and resolves dependencies.

Package dependencies are defined using a combination of the package *name* and *version range descriptor*. This descriptor is specified with a simple dedicated syntax, that extends some of the standard versioning notation. Below a snippet taken from the `package.json` file in the NPM manual:

```
"dependencies" : {
   "foo" : "1.0.0 - 2.9999.9999",
   "bar" : ">=1.0.2 <2.1.2",
   "baz" : ">1.0.2 <=2.3.4",
   "boo" : "2.0.1",
   "qux" : "<1.0.0 || >=2.3.1 <2.4.5",
   "asd" : "http://asdf.com/asdf.tar.gz",
   "til" : "~1.2",
   "elf" : "~1.2.3",
   "two" : "2.x",
   "thr" : "3.3.x",
}
```

The version range descriptor syntax is a powerful tool to specify which version(s) or version range(s) of dependencies are required. It provides the exact information needed to build, install and/or load the software. In contrast to R, NPM takes full advantage of this information. In R, all packages are installed in one or more global libraries, and at any given time a subset of these packages is loaded in memory. This is where NPM takes a very different approach. During installation of a package, NPM creates a *subdirectory* for dependencies inside the installation directory of the package. It compares the list of dependency declarations from the `package.json` with an index of the repository archive, and then constructs a private library containing the full dependency tree and precise versions as specified by the author. Hence, every installed package has its own library of dependencies. This works recursively, i.e. every dependency package inside the library again has its own dependency library.

```
jeroen@ubuntu:~/Desktop$ npm install d3
jeroen@ubuntu:~/Desktop$ npm list
/home/jeroen/Desktop
 d3@2.10.3
   jsdom@0.2.14
    contextify@0.1.3
     bindings@1.0.0
    cssom@0.2.5
    htmlparser@1.7.6
    request@2.12.0
      form-data@0.0.3
       async@0.1.9
       combined-stream@0.0.3
         delayed-stream@0.0.5
      mime@1.2.7
   sizzle@1.1.0
```

By default, a package loads dependencies from its private library, and the namespace of the dependency is imported explicitly in the code. This way, an installed NPM package is completely unaffected by other applications, packages, and package updates being installed on the machine. The private library of any package contains all required dependencies, with the exact versions that were used to develop the package. A package or application that has been tested to work with certain versions of its dependencies, can easily be installed years later on another machine, even though the latest versions of dependencies have had major changes in the mean time.

5.4.3 Back to R

A similar way of managing packages could be very beneficial to R as well. It would enable the same dynamic development and stable installation of packages that has resulted in a small revolution within the Javascript community. The only serious drawback of this design is that it requires more disk space and slightly more memory, due to multiple versions packages being installed and/or loaded. Yet the memory required to load an additional package is minor in comparison with loading and manipulating a medium sized dataset. Considering the wide availability of low cost disk space and memory these days, we expect that most users and developers will happily pay this small price for more reliable software and reduced debugging time.

Unfortunately, implementing a package manager like NPM for R would require some fundamental changes in the way R installs and loads packages and namespaces, which might break backward compatibility at this point. One change that would probably be required for this is to move away from the `Depends` relation definition, and require all packages to rely on `Imports` and a `NAMESPACE` file to explicitly import objects from other packages. A more challenging problem might be that R should be able to load multiple versions of a package simultaneously while keeping their namespaces separated. This is necessary for example when two packages are in use, which both depend on different versions of one and the same third package. In this case, the objects, methods and classes exported by the dependency package should affect only to the package that imported them.

Finally, it would be great if the package manager was capable of installing multiple versions of a package inside a library, for example by appending the package version to the name of the installation directory (e.g. `MASS_7.3-22`). The `library` and `require` functions could then be extended with an argument specifying the version to be loaded. This argument could use the same version range descriptor syntax that packages use to declare dependencies. Missing versions could automatically be installed, as nothing gets overwritten.

```r
library(ggplot2, version = "0.8.9")
library(MASS, version = "7.3-x")
library(Matrix, version = ">=1.0")
```

Code as above leaves little ambiguity and tremendously increases reliability and reproducibility of R code. When the code is explicit about which package versions are loaded, and packages are explicit about dependency versions, an R script or Sweave document that once worked on a certain version of R, will work for other users, on different systems, and keep working over time, regardless of upstream changes. For users not concerned with dependency versioning, the default value of the `version` argument could be set to `"*"`. This value indicates that any version will do, in which case the package manager gives preference to the most recent available version of the package.

The benefits of a package manager capable of importing specific versions of packages would not just be limited to contributed code. Such a package manager would also reduce the necessity to include all of the standard library and more in the R releases. If implemented, the R Core team could consider moving some of the *base* and *recommended* packages out of the `r-base` distribution, and offer them exclusively through CRAN. This way, the R software could eventually become the minimal core containing only the language interpreter and package manager, similar to e.g. Node and NPM. More high-level functionality could be loaded on demand as versioning is controlled by the package manager. This would allow for less frequent releases of the R software itself, and further improve compatibility and reproducibility between versions of R.

5.5 Summary

The infrastructure for contributed code has supported the steady growth and adoption of the R software. For the majority of users, contributed code is just as essential in their daily work as the R base software suite. But the number of packages on CRAN has grown beyond what could have been foreseen, and practices and policies that used to work on a smaller scale are becoming unsustainable. At the same time there is an increasing demand for more reliable, stable software, that can be used as part of embedded systems, enterprise applications, or reproducible research. The design and policies of CRAN and the package manager shape the development process and play an important role in determining the future of the platform. The current practice of publishing package updates directly to end-users facilitates a highly versatile development, but comes at the cost of reliability. The default behavior of R to install packages in a single library with only the latest versions is perhaps more appropriate for developers than regular users. After nearly two decades of development, R has reached a maturity where a slightly more conservative approach could be beneficial.

This paper explained the problem of dependency versioning, and tried to make a case for transitioning to a system that does not assume that package versions are interchangeable. The most straightforward approach would be by extending the *r-release* and *r-devel* branches to the full CRAN repository, and only publish updates of contributed packages to the *r-devel* branch of R. This way, the *stable* versions of R are tied to a fixed version of each CRAN package, making the code base and behavior of a given release of R less ambiguous. Furthermore, a release cycle allows us to concentrate coordination and testing efforts for contributed packages along with releases of R, rather than continuously throughout the year.

In the long term, a more fundamental revision of the packaging system could be considered, in order to facilitate dynamic contributed development without sacrificing reliability. However, this would involve major changes in the way libraries and namespaces are managed. The most challenging problem will be support for concurrently loading multiple versions of a package. But when the time is ready to make the jump to the next major release of R, we hope that R Core will consider revising this important part of the software, adopting modern approaches and best practices of package management that are powering collaboration and uniting efforts within other open source communities.

APPENDIX A

Example profiles

This appendix prints some of the example profiles that ship with the `RAppArmor` package. To load them in `AppArmor`, an ascii file with these rules needs to be copied to the `/etc/apparmor.d/` directory. After adding new profiles to this directory they can be loaded in the kernel by running `sudo service apparmor restart`. The `r-cran-rapparmor` package that can be build on Debian and Ubuntu does this automatically during installation. Once profiles have been loaded in the kernel, any user can apply them to an `R` session using either the `aa_change_profile` or `eval.secure` function from the `RAppArmor` package.

A.1 Profile: `r-base`

```
#include <tunables/global>
profile r-base {
        #include <abstractions/base>
        #include <abstractions/nameservice>

        /bin/* rix,
        /etc/R/ r,
        /etc/R/* r,
        /etc/fonts/** mr,
        /etc/xml/* r,
        /tmp/** rw,
        /usr/bin/* rix,
        /usr/lib/R/bin/* rix,
        /usr/lib{,32,64}/** mr,
        /usr/lib{,32,64}/R/bin/exec/R rix,
        /usr/local/lib/R/** mr,
        /usr/local/share/** mr,
        /usr/share/** mr,
}
```

A.2 Profile: `r-compile`

```
#include <tunables/global>
profile r-compile {
        #include <abstractions/base>
        #include <abstractions/nameservice>

        /bin/* rix,
        /etc/R/ r,
        /etc/R/* r,
        /etc/fonts/** mr,
        /etc/xml/* r,
        /tmp/** rmw,
        /usr/bin/* rix,
        /usr/include/** r,
        /usr/lib/gcc/** rix,
        /usr/lib/R/bin/* rix,
        /usr/lib{,32,64}/** mr,
        /usr/lib{,32,64}/R/bin/exec/R rix,
        /usr/local/lib/R/** mr,
        /usr/local/share/** mr,
        /usr/share/** mr,
}
```

A.3 Profile: `r-user`

```
#include <tunables/global>
profile r-user {
        #include <abstractions/base>
        #include <abstractions/nameservice>

        capability kill,
        capability net_bind_service,
        capability sys_tty_config,

        @{HOME}/ r,
        @{HOME}/R/ r,
        @{HOME}/R/** rw,
        @{HOME}/R/{i686,x86_64}-pc-linux-gnu-library/** mrwix,
        /bin/* rix,
        /etc/R/ r,
        /etc/R/* r,
        /etc/fonts/** mr,
        /etc/xml/* r,
        /tmp/** mrwix,
        /usr/bin/* rix,
        /usr/include/** r,
        /usr/lib/gcc/** rix,
        /usr/lib/R/bin/* rix,
        /usr/lib{,32,64}/** mr,
        /usr/lib{,32,64}/R/bin/exec/R rix,
        /usr/local/lib/R/** mr,
        /usr/local/share/** mr,
        /usr/share/** mr,
}
```

APPENDIX B

Security unit tests

This appendix prints a number of unit tests that contain malicious code and which should be prevented by any sandboxing tool.

B.1 Access system files

Usually `R` has no business in the system logs, and these are not included in the profiles. The codechunk below attempts to read the syslog file.

```
readSyslog <- function() {
    readLines("/var/log/syslog")
}
```

When executing this with the r-user profile, access to this file is denied, resulting in an error:

```
eval.secure(readSyslog(), profile = "r-user")
```

B.2 Access personal files

Access to system files can to some extend by prevented by running processes as non privileged users. But it is easy to forget that also the user's personal files can contain senstive information. Below a simple function that scans the `Documents` directory of the current user for files containing credit card numbers.

```r
findCreditCards <- function() {
    pattern <- "([0-9]{4}[- ]){3}[0-9]{4}"
    for (filename in dir("~/Documents", full.names = TRUE, recursive = TRUE)) {
        if (file.info(filename)$size > 1e+06)
            next
        doc <- readLines(filename)
        results <- gregexpr(pattern, doc)
        output <- unlist(regmatches(doc, results))
        if (length(output) > 0) {
            cat(paste(filename, ":", output, collapse = "\n"), "\n")
        }
    }
}
```

This example prints the credit card numbers to the console, but it would be just as easy to post them to a server on the internet. For this reason the `r-user` profile denies access to the user's home dir, except for the ~/`R` directory.

B.3 Limiting memory

When a system or service is used by many users at the same time, it is important that we cap the memory that can be used by a single process. The following function generates a large matrix:

```r
memtest <- function() {
    A <- matrix(rnorm(1e+07), 10000)
}
```

When `R` tries to allocate more memory than allowed, it will throw an error:

```r
A <- eval.secure(memtest(), RLIMIT_AS = 50 * 1024 * 1024)

## Error:  cannot allocate vector of size 76.3 Mb
```

```
A <- eval.secure(memtest(), RLIMIT_AS = 500 * 1024 * 1024)

## Warning:  Failed to kill process.  The pid or process group does not exist.

## Error:  R call did not return within 60 seconds.  Terminating process.
```

B.4 Limiting CPU time

Suppose we are hosting a web service and we want to kill jobs that do not finish within 5 seconds. Below is a snippet that will take much more than 5 seconds to complete on most machines. Note that because `R` calling out to `C` code, it will not be possible to terminate this function prematurely using R's `setTimeLimit` or even using `CTRL+C` in an interactive console. If this would happen inside of a bigger system, the entire service might become unresponsive.

```
cputest <- function() {
    A <- matrix(rnorm(1e+07), 1000)
    B <- svd(A)
}
```

In RAppArmor we have actually two different options to deal with this. The first one is setting the `RLIMIT_CPU` value. This will cause the kernel to kill the process after 5 seconds:

```
system.time(x <- eval.secure(cputest(), RLIMIT_CPU = 5))

## Warning:  Failed to kill process.  The pid or process group does not exist.
## Warning:  Failed to kill process.  The pid or process group does not exist.

## Error:  R call did not return within 60 seconds.  Terminating process.

## Timing stopped at: 4.846 0.175 5.102

print(x)

## Error:  object 'x' not found
```

However, this is actually a bit of a harsh measure: because the kernel automatically terminates the process after 5 seconds we have no control over what should happen when this happens, nor can we throw an informative error. Setting `RLIMIT_CPU` is a bit like starting a job with a self-destruction timer. A more elegant solution is to terminate the process from `R` using the `timeout` argument from the `eval.secure` function. Because the actual job is processed in a fork, the parent process stays responsive, and is used to kill the child process.

```
system.time(x <- eval.secure(cputest(), timeout = 5))

## Error:  R call did not return within 5 seconds.  Terminating process.

## Timing stopped at: 0.001 0.002 5.003
```

One could even consider a Double Dutch solution by setting both `timeout` and a slightly higher value for `RLIMIT_CPU`, so that if all else fails, the kernel will end up killing the process and its children.

B.5 Fork bomb

A fork bomb is a process that spawns many child processes, which often results in the operating system getting stuck to a point where it has to be rebooted. Performing a fork bomb in `R` is quite easy and requires no special privileges:

```
forkbomb <- function() {
    repeat {
        parallel::mcparallel(forkbomb())
    }
}
```

Do not call this function outside sandbox, because it will make the machine unresponsive. However, inside our sandbox we can use the `RLIMIT_NPROC` to limit the number of processes the user is allowed to own:

```
eval.secure(forkbomb(), RLIMIT_NPROC = 100)

## Error:  unable to fork, possible reason:  Resource temporarily unavailable
```

Note that the process count is based on the `Linux` user. Hence if the same `Linux` user already has a number of other processes, which is usually the case for non-system users, the cap has to be higher than this number. Also note that in some `Linux` configurations, the root user is exempted from the `RLIMIT_NPROC` limit.

Different processes owned by a single user can enforce different `NPROC` limits, however in the actual process count all active processes from the current user are taken into account. Therefore it might make sense to create a separate Linux system user that is only used to process `R` jobs. That way `RLIMIT_NPROC` actually corresponds to the number of concurrent `R` processes. The `eval.secure` arguments `uid` and `gid` can be used to switch Linux users before evaluating the call. E.g to add a system user in `Linux`, run:

```
sudo useradd testuser --system -U -d/tmp -c"RAppArmor Test User"
```

If the main `R` process has superuser privileges, incoming call can be evaluated as follows:

```
eval.secure(run_job(), uid = "testuser", RLIMIT_NPROC = 10, timeout = 60)
```

Bibliography

A. M. Kuchling. Functional programming. *Python Documentation*, 2014. URL `http://docs.python.org/2/howto/functional.html`. Release 0.31.

M. Abu Rajab, J. Zarfoss, F. Monrose, and A. Terzis. A multifaceted approach to understanding the botnet phenomenon. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 41–52. ACM, 2006. URL `conferences.sigcomm.org/imc/2006/papers/p4-rajab.pdf`.

ACM. ACM honors dr. John M. Chambers of Bell Labs with the 1998 ACM software system award for creating `S` system, 1998. URL `http://www.acm.org/announcements/ss99.html`.

E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide (Software, Environments and Tools)*. Society for Industrial and Applied Mathematics, 3 edition, 1 1987. ISBN 9780898714470. URL `http://amazon.com/o/ASIN/0898714478/`.

M. Armbrust et al. A view of cloud computing. *Communications of the ACM*, 53 (4):50–58, 2010. URL `http://dl.acm.org/citation.cfm?id=1721672`.

Jeff Banfield. `Rweb`: Web-based statistical analysis. *Journal of Statistical Software*, 4(1):1–15, 3 1999. ISSN 1548-7660. URL `http://www.jstatsoft.org/v04/i01`.

D. Bates and D Eddelbuettel. *Using `R` on Debian: Past, Present, and Future*, 2004. URL `http://www.r-project.org/conferences/useR-2004/abstracts/Eddelbuettel+Bates+Gebhardt.pdf`. UseR 2004.

Douglas Bates, Martin Maechler, and Ben Bolker. *`lme4`: Linear Mixed-Effects*

Models Using S4 Classes, 2011. URL `http://CRAN.R-project.org/package=lme4`. R package version 0.999375-39.

M. Bauer. Paranoid Penguin: an Introduction to Novell `AppArmor`. *Linux Journal*, 2006(148):13, 2006. URL `www.linuxjournal.com/article/9036`.

R. A. Becker and J. M. Chambers. *S: An Interactive Environment for Data Analysis and Graphics (His Competencies for Teaching; V. 3)*. Chapman and Hall/CRC, 0 edition, 2 1984. ISBN 9780534033132. URL `http://amazon.com/o/ASIN/053403313X/`.

R. A. Becker, J. M. Chambers, and Allan R Wilks. *New S Language*. Chapman and Hall/CRC, 6 1988. ISBN 9780534091934. URL `http://amazon.com/o/ASIN/0534091938/`.

Alex Bertram. ***Renjin**: JVM-based Interpreter for **R***, 2012. URL `http://code.google.com/p/renjin/`.

André B Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd international workshop on Software and performance*, pages 195–203. ACM, 2000. URL `http://dl.acm.org/citation.cfm?id=350432`.

Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D^3 data-driven documents. *Visualization and Computer Graphics, IEEE Transactions on*, 17(12): 2301–2309, 2011. URL `http://dl.acm.org/citation.cfm?id=2068631`.

David R Brillinger. John W. Tukey: his life and professional contributions. *Annals of Statistics*, pages 1535–1575, 2002. URL `http://www.stat.berkeley.edu/~brill/Papers/life.pdf`.

Canonical, Inc. *Ubuntu 12.04 Precise Manual: GETRLIMIT(2)*, 2012. URL `http://manpages.ubuntu.com/manpages/precise/man2/getrlimit.2.html`.

Kai Chew. *Cloudstat: Analyze Big Data With* `R` *in the Cloud*, 2012. URL `http://www.cloudstat.org`.

Kristina Chodorow. *MongoDB: The Definitive Guide*. O'Reilly Media, second edition edition, 5 2013. ISBN 9781449344689. URL `http://amazon.com/o/ASIN/1449344682/`.

James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

Alex Couture-Beil. *rjson: JSON for R*, 2013. URL `http://CRAN.R-project.org/package=rjson`. R package version 0.2.13.

B.J. Cox. Planning the software industrial revolution. *Software, IEEE*, 7(6):25–33, Nov 1990. ISSN 0740-7459. doi: 10.1109/52.60587. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=60587`.

D. Crockford. The `application/json` Media Type for JavaScript Object Notation (`JSON`). RFC 4627 (Informational), July 2006a. URL `http://www.ietf.org/rfc/rfc4627.txt`. Obsoleted by RFCs 7158, 7159.

Douglas Crockford. JSON: The fat-free alternative to XML. In *Proc. of XML*, volume 2006, 2006b. URL `http://www.json.org/fatfree.html`.

L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. Social coding in github: Transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pages 1277–1286. ACM, 2012. URL `http://dl.acm.org/citation.cfm?id=2145396`.

David B. Dahl and Scott Crawford. Rinruby: Accessing the r interpreter from pure ruby. *Journal of Statistical Software*, 29(4):1–18, 1 2009. ISSN 1548-7660. URL `http://www.jstatsoft.org/v29/i04`.

Gergely Daroczi. *The `sandboxR` package: Filtering "malicious" Calls in `R`*, 2013. URL `https://github.com/Rapporter/sandboxR`.

Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, January 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492. URL `http://dx.doi.org/10.1145/1327452.1327492`.

Mike Dewar. First steps in data visualisation using d3.js, 2012. URL `http://vimeo.com/35005701#t=7m17s`. New York Open Statistical Programming Meetup on Jan. 12, 2012.

Ecma International. ECMAScript Language Specification. *European Association for Standardizing Information and Communication Systems*, 1999. URL `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`.

Dirk Eddelbuettel and Charles Blundell. cran2deb: A fully automated cran to debian package generation system. Presented at UseR Conference, July 10-12, Rennes, 2009. URL `https://r-forge.r-project.org/projects/cran2deb/`.

Dirk Eddelbuettel and Romain Francois. *`RInside`: `C++` Classes to Embed `R` in `C++` Applications*, 2011a. URL `http://CRAN.R-project.org/package=RInside`. R package version 0.2.4.

Dirk Eddelbuettel and Romain Francois. Rcpp: Seamless r and c++ integration. *Journal of Statistical Software*, 40(8):1–18, 4 2011b. ISSN 1548-7660. URL `http://www.jstatsoft.org/v40/i08`.

Dirk Eddelbuettel, Murray Stokely, and Jeroen Ooms. `RProtoBuf`: Efficient Cross-Language Data Serialization in R. *arXiv:1401.7372*. URL `http://arxiv.org/abs/1401.7372`.

Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 1 edition, 8 2003. ISBN 9780321125217. URL `http://amazon.com/o/ASIN/0321125215/`.

R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. URL `http://www.ietf.org/rfc/rfc2616.txt`. Updated by RFCs 2817, 5785, 6266, 6585.

Roy T Fielding. Rest apis must be hypertext-driven. *Untangled musings of Roy T. Fielding*, 2008. URL `http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven`.

Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000. URL `https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`. AAI9980887.

J. Fox. Aspects of the Social Organization and Trajectory of the R Project. *The R Journal*, 1(2):5–13, 2009. URL `http://journal.r-project.org/archive/2009-2/RJournal_2009-2_Fox.pdf`.

John Franks, P Hallam-Baker, J Hostetler, S Lawrence, P Leach, Ari Luotonen, and L Stewart. RFC 2617: HTTP Authentication: Basic and Digest Access Authentication, 1999. URL `https://tools.ietf.org/html/rfc2617`.

Free Software Foundation. *GETRLIMIT – Linux Programmer's Manual*, 2012. URL `http://www.kernel.org/doc/man-pages/online/pages/man2/setrlimit.2.html`.

Sadayuki Furuhashi. *MessagePack: It's like JSON. but fast and small*, 2014. URL `http://msgpack.org/`.

F. Galiegue and K. Zyp. JSON Schema: core definitions and terminology. *draft-zyp-json-schema-04 (work in progress)*, 2013. URL `https://tools.ietf.org/html/draft-zyp-json-schema-04`.

L Gautier. *rpy2: A simple and efficient access to R from Python*, 2012. URL `http://rpy.sourceforge.net/rpy2.html`.

R.C. Gentleman, V.J. Carey, D.M. Bates, B. Bolstad, M. Dettling, S. Dudoit, B. Ellis, L. Gautier, Y. Ge, J. Gentry, et al. Bioconductor: open software development for computational biology and bioinformatics. *Genome biology*, 5 (10):R80, 2004. URL `http://www.ncbi.nlm.nih.gov/pubmed/15461798`.

T. Harada, T. Horie, and K. Tanaka. Task Oriented Management Obviates Your Onus on `Linux`. In *Linux Conference*, 2004. URL `http://sourceforge.jp/projects/tomoyo/document/lc2004-en.pdf`.

D. Hardt. The OAuth 2.0 Authorization Framework. RFC 6749 (Proposed Standard), October 2012. URL `http://www.ietf.org/rfc/rfc6749.txt`.

Richard M. Heiberger and Erich Neuwirth. *R Through Excel: A Spreadsheet Interface for Statistics, Data Analysis, and Graphics (Use R!)*. Springer, 2009 edition, 8 2009. ISBN 9781441900517. URL `http://amazon.com/o/ASIN/1441900519/`.

George T. Heineman and William T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley Professional, 6 2001. ISBN 9780201704853. URL `http://amazon.com/o/ASIN/0201704854/`.

John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, 5 edition, 10 2011. URL `http://amazon.com/o/ASIN/B0067KU84U/`.

J. Horner and D Eddelbuettel. *littler: a scripting front-end for GNU R. littler version 0.1.5*, 2011. URL `http://dirk.eddelbuettel.com/code/littler.html`.

Jeffrey Horner. ***rApache**: Web Application Development with **R** and Apache.*, 2011. URL `http://www.rapache.net/`.

Jeffrey Horner. *RApache: Web application development with R and Apache*, 2013. URL `http://www.rapache.net`.

Ross Ihaka. ⁝ Past and future history. *Computing Science and Statistics*, pages 392–396, 1998.

IOCCC. The international obfuscated çode contest, 2012. URL `http://www.ioccc.org`.

Max Kuhn. *caret: Classification and Regression Training*, 2013. URL `http://CRAN.R-project.org/package=caret`. R package version 5.16-04.

Max Kuhn. *CRAN Task View: Reproducible Research*, 2014. URL `http://cran.r-project.org/web/views/ReproducibleResearch.html`.

Duncan Temple Lang. ***RJSONIO**: Serialize **R** Objects to JSON, JavaScript Object Notation*, 2012. R package version 0.98-0.

C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979. ISSN 0098-3500. doi: 10.1145/355841.355847. URL `http://doi.acm.org/10.1145/355841.355847`.

Herbert Lee, Bruce Lindsay, Samantha C. Prins, Nicholas P. Jewell, Michelle Dunn, Steven Snapinn, David Banks, and Leonard Stefanski. The future of publication in the statistical sciences. Technical report,

2013. URL `http://magazine.amstat.org/wp-content/uploads/2013an/FuturePublicationsReport.pdf`.

F. Leisch. `Sweave`: Dynamic generation of statistical reports using literate data analysis. *Proceedings of CompStat 2002*, 2002.

James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela H Byers. Big data: The next frontier for innovation, competition, and productivity. *McKinsey Global Institute*, 2011. URL `http://www.mckinsey.com/insights/business_technology/big_data_the_next_frontier_for_innovation`.

J. Mirkovic and P. Reiher. A taxonomy of ddos attack and ddos defense mechanisms. *ACM SIGCOMM Computer Communication Review*, 34(2):39–53, 2004. URL `www.eecis.udel.edu/~sunshine/publications/ccr.pdf`.

W. Moreira and G.R. Warnes. `RPy`: `R` from `Python`, 2006. URL `http://rpy.sourceforge.net/rpy/README`.

Erich Neuwirth and Thomas Baier. Embedding R in standard software, and the other way around. In *Proceedings of the Distributed Statistical Computing 2001 Workshop*, 2001. URL `http://www.r-project.org/conferences/DSC-2001/Proceedings/NeuwirthBaier.pdf`.

Deborah Nolan and Duncan Temple Lang. Computing in the statistics curricula. *The American Statistician*, 64(2), 2010. URL `http://www.stat.berkeley.edu/~statcur/Preprints/ComputingCurric3.pdf`.

Deborah Nolan and Duncan Temple Lang. *XML and Web Technologies for Data Sciences with R*. Springer, 2014. URL `http://link.springer.com/book/10.1007/978-1-4614-7900-0`.

J.C.L. Ooms. *Stockplot web application: A Web Interface for Plotting Historical Stock Values.*, 2009. URL `http://rweb.stat.ucla.edu/stockplot`.

J.C.L. Ooms. *lme4 web application: A Web Interface for the `R` Package `lme4`*, 2010. URL `http://rweb.stat.ucla.edu/lme4`.

Jeroen Ooms. *`OpenCPU`: Producing and Reproducing Results*, 2013. URL `http://www.opencpu.org`.

Jeroen Ooms, Duncan Temple Lang, and Jonathan Wallace. *jsonlite: A smarter JSON encoder/decoder for R*, 2014. URL `http://github.com/jeroenooms/jsonlite#readme`. R package version 0.9.7.

Roger D. Peng. Reproducible Research in Computational Science. *Science*, 334 (6060):1226–1227, December 2011. ISSN 1095-9203. doi: 10.1126/science.1213847. URL `http://dx.doi.org/10.1126/science.1213847`.

R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014a. URL `http://www.R-project.org/`.

R Core Team. Writing r extensions. *R Foundation for Statistical Computing*, 2014b. URL `http://cran.r-project.org/doc/manuals/R-exts.html`.

R Development Core Team. *`R`: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2012. URL `http://www.R-project.org/`. ISBN 3-900051-07-0.

Chris Reade. *Elements Of Functional Programming (International Computer Science Series)*. Addison-Wesley, 1 edition, 1 1989. ISBN 9780201129151. URL `http://amazon.com/o/ASIN/0201129159/`.

Brian Ripley. The ̊development process. The R User Conference 2011, 2011. URL http://web.warwick.ac.uk/statsdept/user2011/invited/user2011_Ripley.pdf.

Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. *Commun. ACM*, 17(7):365–375, July 1974. ISSN 0001-0782. doi: 10.1145/361011.361061. URL http://doi.acm.org/10.1145/361011.361061.

RStudio Inc. *httpuv: HTTP and WebSocket server library*, 2014a. URL http://CRAN.R-project.org/package=httpuv. R package version 1.3.0.

RStudio Inc. *shiny: Web Application Framework for R*, 2014b. URL http://CRAN.R-project.org/package=shiny. R package version 0.9.1.

S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux Security Module. *NAI Labs Report*, 1:43, 2001. URL http://www.nsa.gov/research/_files/publications/implementing_selinux.pdf.

Leonard Stefanski et al. The future of publication in the statistical sciences. *The Membership Magazine of the American Statistical Association*, 2013. URL http://magazine.amstat.org/wp-content/uploads/2013an/FuturePublicationsReport.pdf.

Linus Torvalds and Junio Hamano. Git: Fast version control system. 2010. URL http://git-scm.com.

John W. Tukey. *Exploratory Data Analysis*. Pearson, 1 edition, 1977. ISBN 9780201076165. URL http://amazon.com/o/ASIN/0201076160/.

Simon Urbanek. *JRI - Java- R Interface*, 2011. URL http://www.rforge.net/JRI/index.html. JRI is now part of rJava.

Simon Urbanek. *Rserve: Binary R server*, 2013a. URL http://CRAN.R-project.org/package=Rserve. R package version 0.6-8.1.

Simon Urbanek. *rJava: Low-level R to Java interface*, 2013b. URL `http://CRAN.R-project.org/package=rJava`. R package version 0.9-6.

Simon Urbanek. ***rJava**: Low-Level **R** to **Java** Interface*, 2013c. URL `http://CRAN.R-project.org/package=rJava`. `R` package version 0.9-4.

Simon Urbanek. *Rserve: Binary R server*, 2013d. URL `http://CRAN.R-project.org/package=Rserve`. R package version 1.7-3.

Stef van Buuren and Jeroen CL Ooms. Stage line diagram: An age-conditional reference diagram for tracking development. *Statistics in medicine*, 28(11): 1569–1579, 2009. URL `http://onlinelibrary.wiley.com/doi/10.1002/sim.3567/abstract`.

W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S*. Springer, New York, fourth edition, 2002. URL `http://www.stats.ox.ac.uk/pub/MASS4`. ISBN 0-387-95457-0.

Hadley Wickham. ***ggplot2**: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2009. ISBN 978-0-387-98140-6. URL `http://had.co.nz/ggplot2/book`.

Hadley Wickham. Tidy Data. *Under review*, 2014. URL `http://vita.had.co.nz/papers/tidy-data.pdf`.