Secure Remote Execution for the R Programming Environment

(extended abstract of the MSc dissertation)

Francisco Marques Canelas Ferreira Banha
Departamento de Engenharia Informática
Instituto Superior Técnico
Advisor: Professor João Garcia

Abstract — The R programming language is mainly used for statistical computation and graphics, which often uses a lot of computational resources. R programmers alternate their activity between long periods of low resource consumption when they are editing their code, and long periods of high consumption when they are waiting for their programs to finish executing. While editing their code, R programmers could lend their computer to other programmers, so that they can take advantage of their available computational resources to run their simulations.

In this document we propose a system where R programmers can safely share their computational resources among each other. This will allow programs to be executed much faster, in a distributed manner. We will do this by running the R programs in an isolated environment with less permissions, so that programs may not cause harm to the host.

Keywords — R, Cycle Sharing, Sandbox, Security

I. INTRODUCTION

R [1] is a system used both for statistical computation and graphics. It is not only a programming language, but also an environment, providing high level graphics, interfaces to other languages and debugging tools. The R language is a dialect of S [2], a widespread programming language also used for statistical purposes, developed by John Chambers et al. at the Bell Laboratories in the 1980s, and was heavily influenced by both S and Scheme [3], developed by Steel and Sussman.

R is mainly used for statistical and analytic purposes, with a growing number of users. It is currently the most used tool for data mining [4]. R computations quickly become extremely heavy for one machine alone since they may require many complex mathematical calculations.

The Internet is a place where one may find a huge quantity of computational power for executing parallel tasks. Some of the time the common user’s computer is idle and is capable of executing available jobs.

Programmers that use R frequently develop programs that tend to consume a lot of resources and take a lot of time, whereas when the programmers are editing their source code the computer is idle for most of the time. The sharing of computational resources of idle computers has been addressed in detail [5], in what can be called “cycle sharing”. Cycle sharing is a way for computers to use more resources than those available to them locally, by exploiting an environment where users can share computational resources that are not being used by their machines. However, most systems are designed in an asymmetric way, where users donate their cycles to a project without receiving any service in return. Also, this kind of projects require that donors explicitly coordinate via a centralized server.

However, if an R programmer is willing to share his computational resources with others so that they can launch their tasks on his computer, this programmer should be protected. Security is an important aspect that needs to be addressed, since resource donors should not become vulnerable to attacks because of malicious code or code with bugs.

Our objective is to have R run inside a sandboxed environment, so that the code that is sent by a remote R programmer may not cause harm to the receiver’s machine, whether on purpose or because of a bug.

II. RELATED WORK

A. The R Programming Language

The R [1] programming language is heavily influenced by two existing languages - Becker, Chambers, and Wilks’ S programming language [2] and Steel and Sussman’s Scheme [3]. In terms of syntax it resembles S, but its implementation and semantics come from Scheme.

R programmers often work in the area of statistics and graphics. R is more than a programming language. It is an environment that provides high level graphics, interfaces to other languages and debugging tools. It has been widely used by data miners [4], allowing programmers to do heavy mathematical calculations within their simulations.

R is an interpreted language, which means that there is an interpreter running that evaluates the expressions input by the programmer. And these expressions are what we need to study in order to better understand how to tackle the subject of identifying when and what should be sent to remote machines in order to speed up the result calculations.

B. Cycle Sharing

The Internet has spread across the globe and is accessible by many people in many countries. And when connected, all the computers form a gigantic network of resources that if harnessed could be used to achieve most of the computational tasks incredibly fast. When computers are idle and their
computational resources are lent to other machines to run the tasks of remote users, this is cycle sharing.

Fig. 1 depicts a schematic view of the common architecture of a cycle sharing environment. The clients first register themselves with the project they wish to share resources with, and download the client that will allow them to do so. Then the server accesses the list of resource donors registered, in order to start the communication with them. Finally the server starts sending tasks to the clients and collecting the results.

Since in the R community, computations tend consume a lot of the machine’s resources, cycle might be a viable option in order to increase the performance of this environment.

1) Isolation-based application-oriented access control: sandboxes and virtualization: The donor of their machine’s computational resources should be protected against malicious code, either because the owner of the received code might be a malicious user, or the communication between them has been hijacked by a third ill-intended party. One way to control the way a program accesses the resources of a machine is to have it run inside a controlled environment, where it can only access other elements that are also inside such an environment. Such an environment is commonly called a 'sandbox'.

The normal behavior of sandboxes is so that there is a complete isolation of the processes running inside it, without having any impact on resources outside of the sandbox’s scope. In reality, many sandboxing schemes implement ways for this isolating barrier to be circumvented, so that data can be copy into and out of sandboxes.

Sandboxing can be accomplished through different approaches, which include Operating System Methods, Virtualization [6] [7] [8] [9] [10] [11], Process Wrappers [12] [13] [14], and Binary Re-writing [15]. Typical sandboxes can either implement one or more of these methods to protect the host’s resources from untrusted guest applications.

Since this dissertation’s focus is only to create a secure execution environment in which to run R code, we will have a specific sandboxed environment for that purpose only, taking advantage of some operating system methods and specific control measures to R’s implementation. There is no need to have a complex sandboxing environment that is flexible enough for different applications and so we can focus our attention on security issues specifically related with R and the performance costs of such modifications.

With this dissertation we want to prevent the R code that is sent by remote machines from altering or accessing the local files without permission for such. However we in no way intend to hinder the local user by altering the way he uses his machine to develop his own R code. We have to manually inspect the source code of R itself in order to inject ways for us to make the decision to deny the user code access to some functions whether it is from the local user or from a remote one. This is not exactly binary re-writing because we are neither making the changes in runtime nor are we altering the binary directly. We are however altering it by analyzing the source code and inserting our modifications there.

III. Architecture

In this section we will talk about the requirements that were identified for our project, as well as our implementation of it. We will describe it both in a way that is not bound to any of the current software and hardware available during this project’s conception, and also how we implemented it with the software and hardware in existence.

A. System Requirements

With this dissertation we will focus on developing security mechanisms in order to protect R programmers that wish to be a part of a resource-sharing community as is presented further in this chapter. More specifically, we will protect their machines’ resources, such as primary storage devices (such as the RAM), secondary storage devices (such as the hard disk), the CPU and the network controller. These are the components that we believe that can be targeted by an R program, since they are the ones that might either contain sensitive or private information, or be used to affect the behavior of the machine. After inspecting the source code of R itself(Listing 1) we could identify the several functionalities that available from within an R program.

Listing 1. List of functions that interact with the operating system extracted directly from R’s source code.

```c
/* Functions To Interact with the Operating System */

["file.show", ...
"file.create", ...
"file.remove", ...
"file.rename", ...
"file.append", ...
"file.symlink", ...
"file.link", ...
"file.copy", ...
"list.files", ...
"list.dirs", ...
"file.exists", ...
"file.choose", ...
"file.info", ...
"file.access", ...
"dir.exists", ...
"dir.create", ...
"tempfile", ...
"tempdir", ...
```
Our project must be able to safeguard the volunteer’s machine, which may receive code from different requesters. It will also provide means of connecting a requester to a volunteer, and transmitting R code and data between them, so that we may properly test and ensure our main goal - the security of the volunteer.

We will present the functional and non-functional requirements that were identified during this project’s planning phase.

1) Functional Requirements: In the following list are the functional requirements that we identified as necessary for a security system to be acceptable for the host machine.

1) The results obtained from running an R program with our solution must be the same as if they were run with the unmodified version of R.

2) The environment must be able to stop programs from reading files from the disk, if the user has set the security profile to do so.

3) The environment must be able to stop programs from writing files to the disk, if the user has set the security profile to do so.

4) The environment must be able to stop programs from opening connections with other machines, if the user has set the security profile to do so.

5) The environment must be able to keep programs from using more memory than what was set with the security profile’s interface.

6) The environment must be able to keep programs from using the CPU for more time than what was set with the security profile’s interface.

7) The security profile should be customizable, so that users can choose how conservative they want the system to be.

2) Non-Functional Requirements: Next, we will list the non-functional requirements that we identified.

1) The performance cost of running R code in our modified environment should be kept to minimum.

2) The owner of the machine that is receiving R code from remote machines should not be disturbed when there is code that is running from different sources.

3) The interface of the security profile should be easy for users to modify.

IV. Architecture Overview

A normal R programmer spends most of the time editing the code and sporadically runs CPU-intensive simulations. When the programmer is done with editing the code and wants to run the simulation, it is often beneficial to use the available resources of other programmers to parallelize its execution.

This thesis focuses on the security issue of a much bigger system, represented in Fig. 2, which is a system that takes advantage of the available remote resources from other R programmers’ machines. This system will work in the following way. There will be a module that works as a node directory and a broker. Each programmer willing to lend their machines’ resources to run other programmers’ programs registers himself by contacting the broker system (represented by message 1 from Fig. 2), that keeps the information of the available machines, and handles the credit awarded and spent by each one. Once a programmer is running their program on their machine and the module responsible for deciding to run the code remotely determines that an operation that they are trying to execute can be parallelized, and decides if it is worth doing it remotely or not. It finds and negotiates with remote resource donors how to distribute the task at hand, through the external broker (messages 2 and 3 from Fig. 2). On the donors side, the communication module receives the program’s code and data, and sends it to the execution module, which executes them in their sandboxed environment (message 4 from Fig. 2). This environment is the focus of this thesis. The owner of the machine has a security profile, that is customizable according to their wishes of having a system conservative enough. The security profile serves as an interface between the owner of the machine and the R environment, in which the owner will tell it whether code received from remote machines:

1) Have permissions to read local files or not.

2) Have permissions to write files to disk or not.

3) Have permissions to open connections with other machines besides the one already established with the sender of the code or not.

4) Only have a certain amount of the machine’s memory space to use.

5) Only have a certain amount of time the process can use the CPU before yielding it to other processes.

When the code has finished executing, the results are sent back to the owner’s machine that verifies their correction (message 5 from Fig. 2). After this is done, the external broker rewards the donors with credit taking into account the amount of resource consumption and the correction of the results (messages 6 and 7 from Fig. 2), which they may spend to run their own code on remote machines that are available to share the workload.
The interface will allow the volunteer to decide how conservative the security rules should be. The more conservative the profile is set to, the less permissions the remote code will have when running on his machine, and the safer it will be, but also the less opportunities the volunteer will have to gain credit to run their own code remotely. We will also restrict the access the remote code has to the disk by limiting its workspace.

This work is a proof of concept that focus on the security mechanisms that protect R programmers from malicious remote code or remote code with bugs that could cause harm to their machine.

### A. Design Decisions

Upon starting to work on this project, the latest version of R available was version 3.2.3, and so all the modifications we have made, have been to this version. Ultimately our objective is to establish a partnership with the R developers, so that our work can be included with future versions of R, and also so that there is the highest numbers of users using our solution possible. The more resources available for users to share, the better the whole system should perform.

1) Inside R’s Source Code: In R’s source code there is a data structure called SEXPREC that is used across the entire system. This data structure has several fields that point to others SEXPREC structures, which represent their values, and thus determine the overall representation of the expression. This is how an R expression is represented. The way R’s source code determines the value of an expression or any of its fields is through the use of several macros that make the conversion between an address of an SEXPREC and the relevant value for the program.

We have inserted a function call to our additional code inside the function that evaluates the SEXPREC’s, and that is how our system decides whether an expression should be blocked or be allowed to be evaluated. We use the existing macros to ascertain the value of the SEXPREC to find out what type of expression or function call is being evaluated at a given moment.

There are three types of function calls that are used by R’s source code - closures, specials, and builtins. Closures are the function calls that are accessible to the user during runtime, they can either be already defined by the language’s specification, or defined by the programmer. Specials and builtins are internal to R, and usually they are called from inside closures already defined by the language’s specification. The difference between specials and builtins is that the specials are evaluated immediately after the call, whereas the builtins’ evaluation is postponed.

We intercept the evaluation of these three types of SEXPREC’s inside R’s own interpreter, which allows us to take advantage of the work done by it during the expressions’ evaluation to make our decision. Doing things this way saves us time from having the surround each function we want analyze with a call to a special function of our own that would identify the expression. We can guarantee that this way, after we enter the interpreter’s code, we will catch the function calls and decide what to do.

To enforce the memory and CPU usage limit, we take advantage of a POSIX function call available in the C language, that allows us to do so. The function setrlimit [16] allows processes to set limits on the resources they use. If the machine’s owner sets a limit too low, then it is unlikely that average R programs will be able to conclude their execution before exhausting this limit, which is why we set these limits to use limits in which the unitary value of it is still enough for programs to be able to finish.

The way our system’s users can customize the security profile is through a text file located in the main directory of R. The portion of this file that contains the values that the user can modify is shown in Listing 2. There is a brief explanation at the beginning of the file describing each option. The file is read once by our code when the programmer starts running the process that will wait for remote calls from other programmers. The information is then stored in a data structure that is kept loaded in memory, so that this only has to be done once. We enforce the security options by comparing the expressions that are evaluated with a subset of R’s function calls that interact with the operating system (Listing 1), which are the ones that are ultimately converted into system calls that attempt to complete the functionalities that were set to be blocked. By doing this evaluation at this lowest level possible, we ignore the syntax used by the program, since in the end all expressions use these function calls. The down side is that this means that if the list of low-level functions is ever altered, our code must also be updated. By doing things like this we control the total amount of CPU time and memory available to the process, as well as access to the Internet, but we do not control the rate at which these resources are used, leaving that task to the operating system of the machine.


<table>
<thead>
<tr>
<th>Memory (in kB)</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU (in minutes)</td>
<td>10</td>
</tr>
<tr>
<td>File reading</td>
<td>ON</td>
</tr>
<tr>
<td>File writing</td>
<td>ON</td>
</tr>
<tr>
<td>Network access</td>
<td>ON</td>
</tr>
</tbody>
</table>

The profile is set to be the most conservative in terms of permissions by default, but will have a reasonably high limit to the CPU and memory usage, so that average sized programs
can execute without issues. The limit will depend greatly from one machine to another, but the value we chose is based solely on the machine we have worked on. When the project reaches its final stage, where it will be as described in section IV, it will be necessary to identify exactly what is the average size of what is transferred between machines, so as to set a proper default value.

If the file is not altered, whenever some R code is received from a remote machine, the process running it will not have permissions to access to any files in it, be it to read or write, and will not be able to create new connections to other machines. To allow processes to do that, the owner of the machine must specify in the security profile that those actions are permitted.

2) Communications Wrapper Code: To handle the code transfer between requester and volunteer we will be using already existing R packages: Rserve [17] is used to launch and run the server, and RSclient - which is a package developed specifically to communicate with a Rserve server. For this work we have also developed a wrapper for the RSclient functions, written in R. This package works by launching a new R process that will act as a server, which awaits remote requests. It then runs the received source code by launching yet another process that will run with the imposed restrictions by the local user through the use of our environment. Not only will this work in a separate execution environment from the local program, but it will cease to exist as soon as it has finished running the remote code and returned its results.

If a machine receives a request from a remote source to execute a portion of its code that requires more permissions than those that its owner has set to environment to give, the code will fail in its execution. On the sending side, the user will only see a message that indicates that an error has occurred. We produce this error through the use of a function call available to us from inside the R’s source code - error(“Error message.”)). Because the communication between machines is not the main focus of this thesis, we left this aspect like so. In future work, this aspect will have to be revised so that, either the machine does not attempt to send code to other machines that will not be able to execute it, or it properly describes why it was impossible to execute the user’s code remotely.

Listing 3 shows how the user can start the server process that will listen and wait for remote calls. These expressions can be stored in a file and be executed by using the R function source. Listing 4 shows how the user can send the code that is to be executed remotely to the server. In this portion of code, we are hiding inside the file client_wrapper.R all the code that is responsible for sending the environment and the expressions, as well as making sure the returned results are then visible to the local user.

Listing 3. R code that launches the server that is waiting for remote calls.

```
library(Rserve)
Rserve(args="--RS-enable-control--RS-enable--remote")
```

Listing 4. R code for the client that wraps all the procedures necessary to send local environment and code to be executed remotely.

```
source("client_wrapper.R")
fcall <- remote_handler()
fcall("example.R")
```

Since our purpose with this thesis is to have a secure environment, we did not make the wrapper too complex. The wrapper will contact the localhost on port 6311 by default, but these may be changed by the user. Listing 5 shows the details behind the wrapper. On lines 41 through 62 we have the several steps we take when a machine is sending its code to another, which we will refer to as client and server respectively from now on, even though all machines may perform both roles. First we identify which packages are being used by the client, so that we can make sure the server has all the necessary packages to run the client code - as shown on line 41. Then we load the package RSclient so that we can communicate with Rserve server. Afterwards, we establish a connection, where we send to the server the information of all the packages that are loaded on the client environment, so that they may be loaded on the server side. As stated in the beginning of this paragraph, we have made this step simple, so that we can focus on testing the security aspect. This is done by the function load_libraries on lines 10 through 13. On the next step, we gather all the global variables present in the environment and send them to the server, which is detailed on lines 16 through 20 in function assign_variables. Again, this step was simplified, in the sense that it would be better to identify and send only the variables that will be used in the computation, instead of all the variables in the environment. It is possible to either load the expressions to send from a file - detailed on lines 21 through 26 in function read_file; or to input them directly - as shown in line 46. Finally we send the expressions to the server to be evaluated - detailed in lines 29 through 31. When the server has finished, we ask for the values of all the global variables on the server side, and update the client environment with those values - detailed on lines 34 through 38 in function update_variables. After that, we close the connection and all the information kept on the server is released, as per the specification of Rserve. If in the beginning the package RSclient was not loaded, it will be unloaded, since we needed to load it to communicate with the server and now the user will no longer need it.

We have several distinct ways of transferring the code between requester and volunteer. The first one is within the function load_libraries, in which we simply create locally the expression to load a library - library(package) - and use the function RSserver.eval to send the expression to the volunteer’s machine. What this function does is it sends the expressions to a buffer on the server side, and those expressions will affect all future connections that the server creates but not the current one, which is why we must close the current connection and create a new one afterwards. We must use this method, because the argument to the function library must be exactly the name of the package and can’t be stored in a variable to be discovered dynamically. The second way we use is within the assign_variables function. Here we simply use RSclient’s function RSassign to create each variable that is defined on the requester side, on the volunteer’s environment. The third way is what we use inside
the function update_variables. We use the function RS.assign to assign to a temporary variable the strings that represents all the expressions to be evaluated on the volunteer side. We then convert the strings to R expressions and evaluate them on the volunteer side. We then remove the temporary variable. Finally, the last transfer is done inside update_variables. Much like in assign_variables, in update_variables we request all the variables declared in the environment, and then use the assign function to update the requester’s environment. However, since we are doing this from the requester to the volunteer, we must retrieve both the name and the value of each variable, which is why we must create a function call of the function get to retrieve each variable’s value.

Listing 5. R code that aggregates all actions performed during the communication phase.

```r
remote_handler <- function(host_ip = "localhost", port_number = 6311) {
  connection = NULL
  open_connection = function() {
    connection <<- RS.connect(host = host_ip, port = port_number)
  }
  close_connection = function() {
    RS.close(connection)
  }
  load_libraries = function(
    installed_packages)
  {
    for (package in installed_packages) {
      RS.server.eval(connection, 
        paste("library(" , 
          package , ")"))
    }
    close_connection()
  }
  assign_variables = function() {
    open_connection()
    vars <- ls(.GlobalEnv, all.names = TRUE)
    for (var in vars) {
      RS.assign(connection, var, get(var))
    }
  }
  read_file = function(filename) {
    conn <- file(filename, open = "r")
    lines <- readLines(conn)
    close(conn)
    return(lines)
  }
  evaluate = function(expressions) {
    RS.assign(connection, "tmp__________", 
      expressions)
    RS.eval(connection, eval(parse

  })
  update_variables = function() {
    vars <- RS.eval(connection, ls(.GlobalEnv, all.names = TRUE))
    for (var in vars) {
      tmp <- as.call(list(quote(get), var))
      assign(var, RS.eval( 
        connection, tmp, lazy = FALSE), envir = .GlobalEnv)
    }
  }
  remote_eval = function(is_file = FALSE) {
    installed_packages <- (.packages())
    library(RSclient)
    open_connection()
    load_libraries(installed_packages)
    assign_variables(
      expressions <- if(is_file) 
        read_file(value) else value
    tryCatch(
      { evaluate(expressions)
        update_variables() 
      }, error = function(condition ) {
        message(condition)
        return(NA)
      }, finally = {
        close_connection()
        if (!("RSclient" %in% 
          installed_packages) 
          ) detach("package: 
            RSclient", unload= TRUE)
      }
    )
    return(remote_eval)
  }
}
```

V. Evaluation

In this section we will describe the methodology that was used to evaluate our implementation and discuss the results obtained. We evaluate different aspects of the solution, such as its correctness and performance penalties.
A. Evaluation Goals and Criteria

The R environment that is being used has been modified to not allow breaches of security, and so each expression evaluated will have to pay the extra cost of going through filter, to verify whether it is permitted or not. We set up a security profile, customizable by the user, that our solution interprets at the start of its execution. When the options of the security profile are set to the most conservative level, the test we ran was to verify whether the filter effectively stopped attempts to access the file system, and tried to create a connection with any other machine besides the sender of the program. After that, we changed the security profile and check whether the results would change accordingly.

We also must guarantee that our modification does not cause programs to have behaviors or results that would not be the same if they were being executed with the original R. And so, if the results of running several benchmarks that require many different functionalities from R would differ, this solution could not be considered acceptable.

Finally, the cost of running R programs in our modified version of the R programming environment should be minimal when compared to the original environment. Some other tests that we ran were to check what would be the cost of running code with our modifications to the R source code, since now it will need to make one extra verification.

B. Experimental Settings

The tests were executed in a machine with the configuration specified in Table I. The solution was developed as a modification to R, version 3.2.3. The packages used to transfer the code were Rserve, version 1.8-5, and RSclient, version 0.7-3. Many more packages had to be installed but only because they were Rserve’s and RSclient’s dependencies. Besides SSH that was used to send commands to the machine, during the tests that only required one R user, only the R environment was running. For the tests that required multiple R users, two instances of the R environment were running on the same machine. This allowed us to focus on the cost of the additional functionality, ignoring possible delays caused by the network.

C. Experimental Results

1) Security Profile: We ran some benchmarks to guarantee that our solution is performing as intended, and maintaining a secure environment in which to run R programs from remote sources. The first group of tests done were meant to identify if we could correctly block the function calls that were not supposed to go through given a certain configuration of the security profile, while letting the others go through. And then, by modifying the criteria of what was allowed, verify that the very same function calls were then allowed to be run.

We also tested whether our solution could correctly prevent programs from using more memory or CPU time than what was stipulated by the security profile. The functionality that will be controlled is the following:

1) Reading files from the local machine.
2) Writing files to disk.
3) Using network connections other than the one used to send the results of the remote code.
4) The amount of memory used by the program.
5) How long the programs can use the CPU to run.

The benchmarks we ran to ascertain this property used simple programs that did little more than attempt to write a file when it should not, or try to connect to an IP address when it was not allowed, or use more memory or CPU time than the maximum limit. By using a simple configuration file the user is able to decide whether it wants to allow these programs to be able to have these functionalities. Should the user decide to restrict the programs, it just means that its machine will not be able to fulfill the request of programs that require these functionalities, and will in turn earn less credit to run its programs on other machines.

We ran tests that start a connection with a remote machine, and attempts to create a file and modify it on it, or attempts to read an existing file’s contents, or even modify an existing file, while the security profile was set to block those functionalities. Then the security profile was altered to allow these types of accesses to go through, and the same program was ran. After that we ran a program that, after starting a connection with a remote machine, attempts to start a connection with a different machine, while the security profile is set to block that functionality, and then again when the security was set to allow it.

As was desired, when the tests were run, if the security profile was set to block those calls, the program failed to complete. After the profile was modified, and the tests run again, they terminated successfully. The same was done with a program attempting to establish a connection with another machine, and the result was the same. Thus we can say that the security profile’s reliability is as it was intended.

2) Correctness of R Programs: To test the second aspect of this project, we used a functionality which R already provides, which uses GNU’s Make tool [18] with a special target to run an extensive benchmark to ascertain whether the language’s specification would differ from what it should be if this was being done with the original R.

make check

This test, even though was crucial, was not entirely within our ability to control its functioning, simply because of how R is built. From the observations that we were able to make, during the building process of R, R code is being executed to create other packages to be used by the final executable.

We kept our additional code in a separate file that is included with all the others that are stored in the main source code directory. The calls to our functions are inserted in existing files written in C code, and that is how we make the bridge between our code and the original R’s. Even though not all
tests from R’s benchmark were terminated successfully, the only differences to the successful answer are values that are dependent on external factors, and thus not deterministic. By simply including this file during the compilation but without having our functions inserted in the original source code, the results of running the **make check** command already differ from the original R version. Without any modification to the original R, the tests for the package **grDevices** already fail. After including our file in the building process, but still without making any calls to our additional code, the tests for the package **base** also fail, where some errors that are mentioned are a difference in an environment variable, and the name of a temporary directory in /tmp. When we include the function calls to our code, the tests for the package **parallel** fail as well, specifically when the test attempts to retrieve the results of an asynchronously evaluated expression from a different process. Because of that we can state that the result of our modifications does not cause deterministic R programs to produce different results.

3) **Performance Cost:** Finally the last test consisted of running several examples in three different modes of R - the original R, without any modification; the modified version of R but without transferring the programs; and the modified version of R, using the remote capabilities to transfer it, to identify the cost of communication. The examples that were used are R programs that require heavy computations, which is something R programmers commonly work with. Be it calculating the determinant of a 2500 x 2500 random matrix, or the calculating 3,500,000 Fibonacci numbers, to the sorting of 7,000,000 random values. Each example was executed multiple times and the average time was calculated.

This test was only be executed on a single machine, using multiple processes when needed. Because the machine will not vary, the time the programs are running should not differ because of hardware differences or network communication times. This will allow us to accurately measure the cost of the modifications introduced in R without concerning ourselves with external factors.

It is expected that running R programs in our modified environment would cause programs to spend some extra time verifying whether the code being executed is from a remote source or from the local user. Besides, when the code is indeed from a remote machine, the instructions have to be intercepted to verify whether they are supposed to be blocked or not.

On Fig. 3, we compare the performance of each type of execution - running the benchmarks with the original R; running the benchmarks with our modified version of R, in a local environment; running the benchmarks with our modified version of R and sending code to remote environments. It shows that both the cost of making the extra verification and the interception of the instructions is not very noticeable. On Fig. 4 we show the same results but normalizing the values to better compare the different versions of R. The worst in which the cost exceeds the double of the original cost is in test T12, which is the creation of a 3000 x 3000 Hilbert matrix. This is likely due to the fact that Hilbert matrices are what can be considered an ill-conditioned matrix, making them especially problematic in numeric computation [19].

These tests can be grouped in three categories: T1 - T5 are tests that do several types of calculations with matrices (like the creation, transposition, or sorting of matrices); T6 - T10 are tests that iterate over matrices applying several functions that either modifies its values (such as inverting the matrices) or determines a certain value representative of the matrices (such as calculating the determinant); T11 - T15 that execute several programming scenarios over a large data structures (like the calculation of a Fibonacci sequence of 3,500,000 elements or calculation the grand common divisors of 400,000 pairs).

---

**VI. CONCLUSIONS**

With this dissertation we hope to have conveyed the idea that there is a great potential for a solution that will allow programmers to increase their efficiency when programming in R. R programmers spend a lot of time editing their code, which leaves their computer with a lot of idle time since the time it takes for a programmer to enter a character is many times slower than the speed that the computer takes to process it. This leaves their computational resources available for other
programmers that have finished editing their code to run their simulations on.

The users of such a system must however be guaranteed that their machines will not be compromised by being a part of a community of programmers that share the idle resources of their machines with each other.

With this work we have developed a proof-of-concept that shows that this kind of solution is possible without leaving the computers vulnerable to malicious or buggy code, and show that R programmers could benefit a lot from using such a solution.

VII. FUTURE WORK

In the future the remaining modules of the final solution must also be developed, so as to perform the tasks that are described in Section IV. In the future there should also be mechanisms that allow a better control of the rate of consumption of the machines’ resources, which should be simpler to integrate with the final module that performs the transmission, and more specifically the reception, of remote code, since there will be less restrictions to its implementation than in our case, in which we have modified an already existing and constantly evolving project - the R Project. The simple implementation of our wrapper code, that transmits the code between machines that was developed so that we could evaluate our solution, is too simplistic for the final architecture. It may serve as an inspiration for future work, but it should be replaced with a more complete solution.

REFERENCES