Bachelor Informatica
Universiteit van Amsterdam

Securing Desktop Grids

Merlijn Wajer

August 2012

Supervisors: Guido van 't Noordende
Signed: Guido van 't Noordende, Dick van Albada

Abstract

Computer grids play an important role in projects which require a relatively large amount of processing power. Traditionally, grids consist of professionally managed cluster computers which run any application provided by authorized users.

It is important that the grid nodes are well protected from malware. The programs run on current desktop-based computer grids are examined manually for possible problematic code and exploits which could compromise the grid.

Manually examining every program is a vulnerable process. A way to protect grid nodes from harmful applications is to run these applications in a so called *jail*. This *jail* monitors the behaviour of each application and prevents the application from executing any harmful commands.

However, this approach requires the jail to have fine tuned knowledge of what is *harmful* and what is not. This project will investigate how transparent a jail can be made, to prevent interference with the programs being run without sacrificing security or performance.

Contents

Chapter 1

Introduction

Scientific research often requires computationally intensive simulation and various other cpu intensive tasks. Researchers do not always have access to the required computational resources to perform their simulations or calculations. Acquiring the required computational resources can be quite expensive and researchers may find themselves unable to acquire (or purchase) these computational resources.

An increasingly popular solution to the growing need for computational resources is a computational grid.

1.1 Grids

A grid is a distributed (networked) system with a non interactive workload [1]. Grids consist of a number of nodes (computers) and the hardware and software amongst the nodes typically varies; grids are often heterogeneous. Grids are a form of distributed computing in which all of the nodes work together to handle computationally intensive jobs.

Other forms of grids rely on communication between running jobs; these grids run interactive applications.

Due to grids typically being owned by the same set of organisations and used by an (assumed) trustworthy group of users, the focus on security implications of running the programs on the grid nodes has never been particularly high.

1.1.1 Desktop Grids and Public Resource Grids

A *Desktop Grid* is a form of distributed computing in which one or several organisations - university, business or other form of organisation - utilise their existing (desktop) computers to handle (long-running) computational tasks.

Public Resource computing (also known as *Volunteer Computing*) is a form of distributed computing very similar to *Desktop Grids* in which *volunteers* donate computer resources to one or more projects. [And04]

Desktop Grids are owned and administered by an organisation and the desktops in the Desktop Grid are often on the same network. The desktops in a Desktop Grid are *trusted*, in the sense that they will not purposely return false

[1] Tasks run without requiring to communicate with the user

computational results, this in contrast to *Public Resource* computing in which the *volunteer* can not be trusted to return correct results; anyone can become a volunteer and it is unwise to trust everyone. Tasks in Public Resource Grids are always run several times on different volunteer computers to ensure the validity of the results, as with SETI@Home[**?**].

Even though Desktop Grids differ from Public Resource Grids in several key areas, the differences are not relevant to this research as both types require extensive security to protect the node from harmful programs. We only focus on the security aspects of running untrusted programs.

Grids do not have to be owned by a single organisation - a lot of organisations also link their computers to create a single large grid.

1.1.2 Cluster Computing

Cluster Computing refers to a group of linked computers. A clusters is owned by a single organisation and managed by IT professionals. The nodes in clusters are usually always available for processing.

This in comparison to most the nodes in Desktop Grids and especially Public Resource Grids in which nodes are not dedicated nodes, this means that they may not be available most of the time - they may be used for other purposes or simply be turned off as the owner of the node seems fit.

Securing nodes in Public Resource Grids therefore is of great importance. Securing nodes is also something that has only been implemented in a somewhat limited fashion in current grid implementations.

1.2 Popular Grid Implementations

Several open source grid systems exist, amongst them are BOINC[And04] and Xtremweb[FGNC00]. These two systems, due to their open nature have become popular grid solutions. This research focusses primarily on desktop grids; when the term grid is used we refer to a desktop grid unless explicitly mentioned otherwise.

1.2.1 BOINC

BOINC [And04](Berkeley Open Infrastructure for Network Computing) was initially developed for use in the SETI@home [ACK+02] project. The first version was not designed with a high emphasis on security and some participants tried to "cheat" on the statistics. BOINC has been designed to solve these issues.

BOINC includes a graphical client user interface to quickly manage several settings as well as the projects the client is connected to. BOINC also includes a *credit system* [And04] which is designed to combat cheating by verifying results, this combats falsification of results as well as handing out more *credits* to participants than they originally deserve.

1.2.2 Xtremweb

Xtremweb is a *Generic Global Computing System* designed in 2000 and presented in [FGNC00]. Xtremweb was designed with two essential features in

mind: delivering high performance, and running multiple applications allowing institutions and research groups to set up their own Global Computing applications. High performance is ensured by efficient scheduling, fault tolerance, striving for scalability and - of course - a large base of nodes.

1.3 Security

The Xtremweb paper states the following:

> *Security* All participating computers should be protected against malicious or erroneous manipulations, and the result of the global computation should not be exposed to be tampered with.

However, the only references in [FGNC00] found related to jailing and sandboxing suggest that it is only provided by means of language sandboxing - a set of rules combined with a safe runtime for a specific programming language to prevent for example, access to certain directories. Such as the Java Virtual Machine (JVM).

Language sandboxing is not fit for running all kinds of different applications.

In contrast to a jail, where the application is monitored on a low (system call) level, which allows any program to run in an isolated environment, regardless of the programming language it was written in.

To our knowledge, current security measures taken by grids [2] are all server-side; except for the possible exemption of running the actual grid client as a user with limited permissions such as the *nobody* user account.

The nobody account is the conventional username of a user account that owns no files and is in no privileged UNIX groups. [3]

Moreover, even with the limited permissions granted to the nobody account it is still possible to seriously harm or compromise a grid node.

BOINC features a *sandbox* [4]: A sandbox is a security mechanism used to virtualise (amongst other things) file access of untrusted programs. By disallowing access to certain files, system calls or other mechanisms a sandbox is often used to execute untested or untrusted code.

But the BOINC sandbox does not appear to be a sandbox in the classical sense; it does not monitor the application's system calls nor does it virtualise the file system or network. It simply ensures that BOINC runs as a user with very limited permissions. The downside is that application running as the limited user can theoretically still compromise the system by exploiting vulnerabilities of software running on the host operating system. To effectively change to a lesser privileged user, the BOINC client requires a method to temporarily elevate its privileges, using *setuid(2)*.

A jailer is a program that creates a jail and imprisons a program in aforementioned jail. Compared to a jailer BOINC's method is quite cumbersome as a

[2] The BOINC client does not provide a jailing mechanism.

[3] Usage of the "nobody" account for daemons is generally discouraged as it is safer to give each daemon its own user with limited permissions, as recommended by the [lin]. If all daemons ran as the nobody user, compromising one daemon would be enough to gain control over all the other daemons as well using the ability to send signals other nobody processes, as well as ptrace other processes owned by nobody.

[4] https://boinc.berkeley.edu/sandbox.php

jailer does not require any elevated privileges, the usage of a jailer is simpler and a jailer implements a more secure jailing mechanism by virtualising file system and other interfaces.

Security measures taken differ per grid project, but usually boil down to the following: **All the applications in the grid project must be verified, either by means of manual code verification or automated auditing.**

Usually, the code is manually verified. While this approach seems quite secure, it does pose some serious problems. The most obvious problem is that the people who verify code can make mistakes. The other - much more important - problem is that the code checking approach does not scale. It is infeasible to validate the code of large numbers of binaries and the libraries they depend on - if the code is available at all, that is. Bugs are found even in well-maintained and often-used commercial code every day - such as in the GNU/Linux operating system - how can Desktop Grid server administrators be expected to verify thousands of lines of third-party code, some of which is rarely used outside the scientific circles? In this case, the grid server administrator can do nothing but trust the author(s) of the code and the user who submits the code.

Manually validating code does not scale as the number of authors increases for large-scale usage of the Desktop grid. Grids are supposed to scale linearly and the current verification model seriously hinders the linear scalability of grids in practice.

Another, less realistic, problem is that the grid server distributing the applications can become compromised and distribute malicious applications to the nodes (clients). If a server however distributes malicious applications when there is no client-side security mechanism in place, clients can be seriously damaged or even compromised.

This paper presents a possible solution to these security issues. It makes use of a jailing system that was initially designed for the Mansion mobile agent system [tNBH+07], [vNBTb], [NOT+]; confinement is used for host protection and to guard confidential information [vNBTa]. Although designed to be portable, the Mansion jailing implementation ran only on 32 bit linux [tNBH+07]. It has been re-implemented for 32 and 64 bit linux [Zup10]; the latter implementation is used in this thesis.

1.4 Jailer

Every program sent to the client will be a **prisoner** for the **jailer**, which should prevent any harm from the applications.

While the jailer itself is fully functional, it has not been tested with a wide variety of applications and needs a proper set of *rules*.

To effectively and securely jail programs with a jailer; a proper set of rules or access policy is required. What files can readily be accessed by any prisoner and what files should the prisoner have no access to are questions answered by the access policy.

To allow the delegated program to run without permissions issues, while at the same time preserving the safety of the client. Providing a working policy for most Public Resource grid applications is the main focus of this research.

1.5 Research question

We will research the viability of running grid applications in a secure environment: a jail. We will investigate how we can transparently imprison applications: applications should still run in the prison and functionality should not be affected; in other words: what is a policy in which most if not all grid programs can run functionally correct?

The purpose of this research is to create a policy that will allow most (if not all) programs to function in the jail while retaining the certainty that the prisoner will not be able to escape the jail.

In Chapter 2 describe a BOINC, Desktop Grid implementation. We will imprison applications from BOINC to test our jailer.

In Chapter 3 we will describe the jailer that is being used to jail BOINC applications.

The performance impact of imprisoning applications is an important aspect; the jailer should not slow down the prisoners too much. We research the performance impact of the jailer on the prisoners in as well as the functionality under the default jailing policy in Chapter 4.

Chapter 2

BOINC

BOINC (Berkeley Open Infrastructure for Network Computing) [And04] is a software system created to manage and create Public Resource computing projects. Amongst goals that BOINC strives to solve are:

- Reducing the barriers to enter Public Resource computing
- Supporting a large range of diverse applications
- Integrating a reward system to reward participants

BOINC is used in this research to retrieve grid programs to run in the jailer. The BOINC framework can be separated into a few components, most importantly the project, the server and the client.

2.1 Project

A BOINC *project* corresponds to an organisation or research group that makes use of public-resource computing. A project consists of a "Master URL", which is the homepage of the project as well as the URL of the scheduling servers. Within a project, several *applications* can exist and these can vary and change over time.

2.2 Server

Every BOINC project needs a *server*. These servers consisting of a set of web services and daemon processes.

Scheduling servers issue work and handle reports of complete results of work done by clients.

Data servers take care of the file uploading by means of certificate based security to ensure that only valid, legitimate files can be uploaded.[1]

All applications downloads are simply done with a web server, over HTTP.

[1] Unless the private key is compromised

2.3 Client

The BOINC *client* is run by the participant and performs all tasks required on the node such as fetching applications from the project servers, running the actual applications and sending back the results. It also features workload management to ensure that BOINC does not use the computer resources when they are required for other (more important) tasks. To fetch projects from the servers the client receives XML files from server with respectively download locations of applications and their information.

The client can operate in different modes: It can run in a *screensaver mode* in which is shows the graphics of the applications that are ran in the node, it can run as an application which provides a tabular view of workload, file transfers and disk usage. It can also run as a Windows service or UNIX command line program.

A project is joined by visiting the project website and registering at the site. If the participant does not have the BOINC client yet, he or she will have to download this as well. After the registration is completed, a "key" is presented to the participant which he or she can use to join the project using the BOINC client.

2.4 Verifying results

BOINC supports *Redundant Computing*, designed to combat possible malicious participants trying to falsify results or possibly malfunctioning hardware (such as faulty memory).

To prevent this from happening, projects can specify that N results should be created for each workunit. Once enough results have been collected, an application-specific function is used to compare the results and determine a canonical result. If this fails, BOINC creates new results for the workunit, this is repeated until a canonical result is found or some limit is reached.

Several server-side daemon processes run to verify computational results: [And04]

- The *transitioner* implements the logic as required for redundant computing; new results are generated when required.
- The *validator*, as the name implies, examines results and determines the canonical result.
- The *assimilator* takes care of new canonical results. It usually has an application-specific function which parses the results and inserts it into a science database.
- The *file deleter* simples deletes data that is no longer required from the data servers.

2.5 Code Signing

In the EDGeS[KLM+08] grid, applications are audited and tested before they are allowed to be send to the clients in the grid environment. [edg] This auditing

is done by looking at the system calls made by a program, if it performs certain (possibly malicious system calls), the program is disallowed from entering the grid environment. This approach has certain drawbacks as the program being audited may perform different system calls based on the system the program is being run on, the program could check if it was being traced for example. Security in the EDGeS grid largely boils down to one thing: trust.

In the SZ-TAKI distribution [MGB+07], a project distributes a list of *Trusted Application Developers*, where each *Application Developer* has its own *certificate* which allows the system to verify the authenticity of the application that is to be distributed. Still, this does not withhold any intentional damage to nodes by an *Application Developer* as this certificate based security is simply a measure to ensure the application is compiled by a trusted *Application Developer*; again, the system is based on trust. [AGZ07]

2.6 Resource Management in BOINC

BOINC has a feature to limit the *"CPU Usage"* to a percentage multiple of 10. BOINC achieves this in a simplistic but cross platform way [2]:

What BOINC does is pause the computation for 1 or more seconds, depending on what percentage of use you set it to.

If the percentage is set to 100%, the computation will run all the time. If it set to anything less, it will idle one or more seconds to "compensate" for running constantly in the non-idle seconds. For example:

- 100%: Run all the time.
- 90%: Run for 9 seconds, idle for 1 second.

[2] http://boincfaq.mundayweb.com/index.php?language=1&view=45

Chapter 3

Jailing System

The security issues with grids previously presented were apparent long before grids came into existence; the need to restrict applications is much older.

Previous applications to confine applications are presented in [GWTB96, KW00, AKS98].

In computer science, a jailer is an application that can confine other programs, called *prisoners*. It is a securely separated environment in which untrusted applications can be executed with very limited access to the rest of the system.

Jailers need a way to "follow" the prisoner to perform the required actions to prevent a prisoner from "breaking out" of the jail. Typically this means system calls will have to be intercepted and possibly changed or rejected.

On Linux, several mechanisms to confine applications exist, such as *Linux-VServer*, *Xen*, *OpenVZ*. These mechanisms are however not viable for they either require special (administrative) permissions or require changes to the Linux kernel and are not a jail in the classical sense. They virtualise much more than just one task.

A low level alternative is **ptrace(2)**, a system call often used by debuggers - programs that run and inspect other programs. ptrace in convenient because it runs on most Unix-like systems including FreeBSD and Solaris and also does not require any special permissions to trace programs. Any program can trace another program with ptrace as long as it has the same (or higher, as in the case of root) permissions than the program it wants to trace. Typically, the program that wants to trace another application starts this application itself. In this case permissions are not really issue as they are inherited from the parent, who is going to trace the newly created child process.

The jailer presented in [Zup10] uses ptrace to imprison applications. Typically the jailer itself starts the program, in which case permissions are not really a problem.

Other ways of confining programs include virtualisation and language-based sandboxing. Virtualisation typically requires an entire operating system to be loaded on top of the current operating system. This program is then executed inside this virtual environment. This causes a lot of overhead and can negatively affect performance. Examples of virtualisation systems are the Linux *KVM* module, *Linux VServer* and *Xen*. These three systems are covered in Chapter 5.

Language based sandboxing examples are *Java* and *Tcl* as described in [OLW97].

3.1 ptrace

Several UNIX and UNIX-like systems support the ptrace system call. This system call allows one process to "control" another process - from now on called *tracee*, enabling the controlling process to stop and inspect the *tracee*, as well as writing to its memory. ptrace is mostly used by tools that aid debugging of software such as gdb[gdb]. ptrace provides two ways of controlling a process - either by checking and validating all the system calls made by the process by stopping the process before and after each system call or by stopping the tracee after each instruction. The latter option has a serious effect on the performance of the tracee and is not required in the implementation of our jailer. All information-sensitive operations are achieved with system calls (reading files, sharing information and so forth).

ptrace stops the tracee right before it executes a system call as well as just after it has executed a system call. While the controlling process is inspecting the tracee, the execution of the tracee is paused. (Figure 3.1)

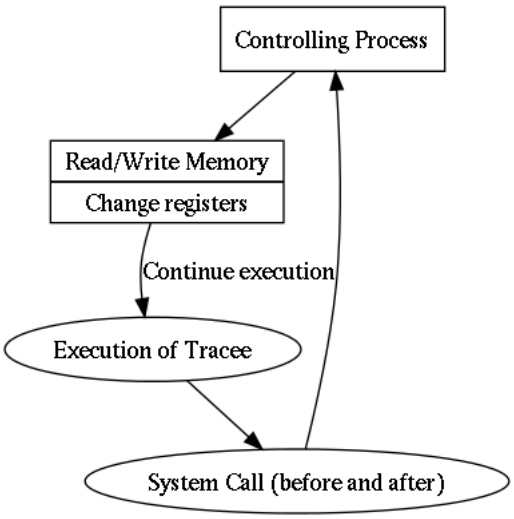


Figure 3.1: ptrace control flow

This gives the controlling process the ability to read and modify registers, which grants the ability to do pretty much anything with the tracee, including but not limited to:

- Modify the system call (number) that is executed.
- Modify the arguments to system calls of the tracee.
- Modify the instruction pointer (commonly called program counter), allowing the controlling process to resume the executing of the tracee at an

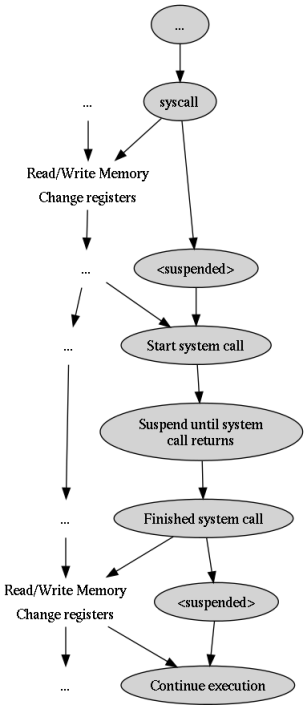


Figure 3.2: Detailed ptrace control flow. Jailer control flow is displayed on the left, prisoner control flow is displayed on the right.

entirely different set of instructions.

- Inject system calls, by modifying the instruction pointer and changing the system call number.
- Modify the return value from the system call.

These features together make jailing processes possible.

3.2 Introduction to the jailer

The jailer presented in [Zup10] uses the ptrace system call to trace prisoners. The prisoner being jailed can perform any action long as it does not interfere with any program running outside the jail. All system calls are intercepted and audited. Because system calls are the only way to interact with the outside world, this is sufficient to confine a process.

Sometimes system calls and the system call arguments have to be changed [1] in order to accept them; this is often the case when the prisoner tries to access files or directories that have been mapped to other directories.

[1] This initially led to race conditions which made it possible to escape a jail. A solution was provided in [tNBH+07] and is implemented in the current jailer.

3.3 Technical background

The code base of the jailer presented in [Zup10] is particularly small and readable which makes it a lot easier to audit the code, in contrast to projects with a large code base. A whitelist approach is preferred over a blacklist as a whitelist approach is inherently more secure by design. A whilelist is used to explicity allow access only to specific files or folders in this case. Conversely, a blacklist is used to explicitly deny access to specific files and folders where all files and folders not on the blacklist are allowed. By default, the jailer denies access to every directory other than the jail directory[2]. Access rules to directories outside the jail have to be explicitly passed to the jailer. Network connections are denied by default as well.

The jailer is divided into several "layers". We will describe these in the sections below.

3.3.1 Trace layer

As previously mentioned, the trace layer uses the special *ptrace* system call. Previous versions of the jailer used a modified version of the *strace* program. To simplify auditing the code, the jailer has its own thin trace layer created specifically to do what the jailer requires and nothing more.

The jailer is also designed to be as system call agnostic as possible, it has no system call specific knowledge, all architecture specific things are abstracted away to keep it as portable as possible.

The layer is stateless to keep it as simple as possible, this does however move the burden of keeping track of what is a pre or post event to another layer: the interception layer. It is necessary to keep track of the state of each process that is traced; because ptrace stops a process right before and after a system call - but it does not provide this information to the program that is tracing processes. The program has to keep track of this itself.

The main function of the trace layer is to intercept system calls and notify the interception layer. It also offers the essential functionality to modify system calls. It can deny system calls; as well as change system call numbers and modify arguments.

3.3.2 Interception layer

The core of the jailer is the interception layer, this layer glues the other layers together: It is invoked by the trace layer and consults the policy layer to determine whether system calls should be allowed or perhaps modified.

Aside from glueing the other two layers together to create a secure environment, the interception layer also manages the resources: Processes, threads, file descriptors and shared read-only memory.

Shared Read-only memory

One of the major problem with ptrace-based jailers was that they suffered from race conditions which allowed the prisoner to escape the jail. The problem in its very essence was that at the moment the system call arguments were verified

[2] –jaildir option in the jailer manpage.

by the jailer, another thread in the prisoner could change the arguments to something that the jailer would have normally rewritten or would have simply denied. This could cause the prisoner to change its arguments and thus break out of the jail and cause harm to the system.

A solution was presented in [tNBH+07] and has been incorporated in this jailer. The solution is illustrated in Figure 3.3. Essentially the jailer copies the system call arguments to the read only memory before checking the arguments, thus ensuring that they will not be changed by the prisoner. The shared read-only memory is writeable by the jailer and read-only for prisoners.

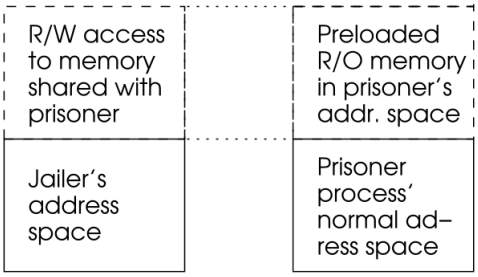


Figure 3.3: Shared Read-only memory

3.3.3 Policy layer

The policy layer determines whether a system call should be allowed (and possibly modified) or denied. It does so based on certain rules passed to the jailer either through command line flags or configuration files. By design, all read and write operations in the jail directory (created by the jailer) are allowed and all the other actions that could affect anything outside the jail are denied.

In chapter 4.3.2 we take a look at what files and directories should and should not be allowed; as well as the several corner cases.

3.4 Deficiencies

By default, the jailer does not allow any kind of networking; both TCP and UDP are disallowed. There is support for allowing and restricting TCP traffic to specific address; with UDP restricting traffic is harder and not fully implemented in the jailer as of yet.

The jailer currently uses the ptrace API to read and write memory from the jailer. A huge downside is that ptrace can only read and write one *processor word* per system call. There are other ways of accessing the memory; such as the */proc/(pid)/mem* file or the *process_vm_readv* and *process_vm_writev* system calls. Both these approaches however require a relatively new Linux kernel.

The jailer also does not use POSIX threads to handle ptrace events; this can become a problem when the jailer is managing a lot of prisoners who are all performing system calls and have to wait for the jailer to verify their system calls.

Chapter 4

Running programs in the jail

4.1 Introduction

The goal of this research is to provide and test the usability of a default policy for the jailer in which most (if not all) trivial parallel programs in BOINC can run functionally correct without being able to cause harm or compromise the system.

To be able to run the computational programs started by BOINC, we had to modify the BOINC source code to access the arguments passed to the computational programs. BOINC downloads these programs and stores them systematically in the BOINC home directory. Since we have access to both the binaries and the arguments passed to the computational program, it is possible to run the program without using BOINC. [1]

To access the parameters passed to the computational programs, we added **fprintf(3)** statements to write all the arguments to a file, see Appendix A.

After acquiring these arguments, it was possible to run the computational programs *standalone*, that is, without having BOINC invoke them. This also makes it easier to use the jailer to run the computational programs. The standalone program reads input from a file, and writes its output to a file after it has completed the computation. Functional correctness can then easily be verified by comparing the output files of the computational program run with and without the jailer.

Performance with and without the jailer will be benchmarked with the *time(1)* command. The performance will be measured without BOINC as BOINC could interfere with the benchmarks by temporarily suspending tasks.

These benchmarks performed with the *time(1)* command are performed to investigate the performance penalty of running the jailer.

We expect that programs run with the jailer should take at least a bit more time than run without the jailer. If the running time with and without the jailer turns out to be very much the same, the jailer hardly has a performance impact.

[1] That is, after having initially used BOINC to download the required computational programs, the input data and pass the arguments.

We do not expect the jailer to have a serious performance impact since most of these program in the grid are CPU-bound and not IO bound - IO generally requires a lot of system calls whereas CPU-bound applications hardly require any system calls. They should (once they have processed all their input data) hardly use any system calls and thus will run without the jailer seriously impacting their performance.

4.2 Jail script and policy

Because the jail denies access to any file or directory unless access is explicitly granted, a sane base environment is typically created before running the prisoner. Provided is a script called *jail*, which sets up aforementioned environment and generates the proper arguments to be passed to the *jailer*, after which it will invoke the jailer.

Programs that are dynamically linked typically need access to the */lib* and */usr/lib* directory; in */bin* and */usr/bin* reside useful applications. Access to these applications is granted since they will also be traced by the jailer, because the prisoner starts the application itself (and all children of the prisoner are traced).

A few other directories are created (*/proc*, */etc*), but access to these is denied, except for a few specific files. For example, */etc/localtime* is a harmless file[2], but other files in /etc/ could contain sensitive information.

Access to these files has to be explicitly passed in the *policy*. Other files, like */etc/passwd* are required by most prisoners as well, but the file itself contains only the user of the prisoner, because the jailer script creates a temporary passwd file.

4.3 Jailer policy

The jailer policy exists of a set of rules that define what folders may be explored and files may be read or even written. The policy is very minimal to minimise the risk of mistakes in the policy.

4.3.1 Policy

The following directories are mapped with the jailer script (with CHROOT=1):

```
/proc
/usr
/bin
/dev
/lib
/lib32
/lib64
/dev
/etc
```

[2] but also very useful!

Note that a mapping alone does now allow any access to these directories.
The default policy with regard to the filesystem is as follows:

```
read and execute:
/usr
/bin
/lib
/lib32
/lib64
```

Apart from the general policy on files an directories, access to the following devices and/or files is permitted:

```
read only access:
/dev/urandom
/etc/localtime

read-write access:
/dev/null
```

4.3.2 Filesystem access

It is useful to consider the effect of adding additional (allowed) paths to policies. Only a few files and folders should be read-write. In almost all cases, normal users do not write to anything but */tmp* and */home* either. Since both these directories are created especially for the prisoner and do not contain any references to or information found in the original directories, it is safe to allow read-write access to these directories.

Dynamically linked applications require the ability to load shared objects (libraries); to avoid cluttering and duplication most shared objects are placed in system directories, typically */lib* and */usr/lib*. It is safe to allow read permissions to all these libraries [3], as all the code in the libraries is traced as well

The shared objects do not contain any sensitive information and they also cannot help the prisoner escape the jail.[4]

Allowing access to some directories or files however can be dangerous; and one should consider the effects of giving access to extra files and directories.

/proc is a pseudo file system which is used as an interface to kernel data structures [man]. This is a filesystem that the prisoner should not be able to access; as it contains a lot of information about other tasks as well as information about the prisoner itself; such as the real current working directory: */proc/self/cwd*.

[3] Obviously, write access is superflous and poses a security risk when the prisoner (and jailer) is run as root user - something that should be avoided regardless.

[4] Remember that all the system calls of a prisoner are traced; giving access to extra libraries means that the prisoner will execute code from these libraries.

Another special directory (or filesystem) is */dev*, which contains special files that are used to interact with devices or provide some other useful functionality. Amongst these are special files like */dev/null*, */dev/urandom* and */dev/zero*, these do not interact with devices attached to the computer; access to at least two of the three previously mentioned files is usually required by applications. Most special files in */dev* should not be accessible for prisoners, they do not require access to them.

Depending on the application of the prisoner, it may be required to allow access to more devices in */dev*, but this is highly application-specific and access should generally be disallowed.

4.4 Applications

To research the performance impact of the jailer as well as test the functional correctness we have written a few tests of our own, as well as three programs that were downloaded to our machine by the BOINC client.

Python: Hashing a file

This program, as seen in Appendix B calculates the hash of a given file N times. Every time it reads the file again; to cause some extra IO load. The block size is configurable; as is the hash algorithm. All the hash function supported by the python *hashlib* module are supported:

```
>>> hashlib.algorithms
('md5', 'sha1', 'sha224', 'sha256', 'sha384', 'sha512')
```

Python: Hashing a file with multiprocessing

This program, as seen in Appendix C, is similar to the previously introduced hash program, but this program makes use of all the cpu cores available. Therefore the theoretical speedup equals the amount of cpus.

Python: Calculating prime numbers

This program, as seen in Appendix D calculates the first N prime numbers and writes the result to a file.

BOINC: Fusion

The Fusion application is a port of the original ISDEP[Fer10] application a grid environment[RBM+]. The ISDEP (Integrator of Stochastic Differential Equations in Plasmas) application aims to solve the dynamics of a fusion plasma (Ionized gas).

The equations that govern the plasma are extremely difficult to solve, as the equations are nonlinear in partial derivative in many dimensions.

BOINC: Search

Search was distributed to us by the BOINC client, but we have not been able to find any specific details about the exact calculations that is performs on the internet. Search is included because the exact calculation performed by the application is not relevant to our research.

BOINC: Correlizer

Correlizer is a program which helps explore and understand the human genomes, it does so by trying to find correlations in the sequential organisation of genomes [cor]:

> The sequential organization of genomes (the relations between distant base pairs and regions within sequences) and its connection to the three-dimensional architectural organization of genomes is still a largely unresolved problem.

4.5 Results

The jailer should not affect the functionality of the program; that is, the results of the computations must not differ if the program is being run within the jail.

Unless mentioned otherwise, the default policy as shown in Policy Layer (Section 4.3.1) is used.

We will measure the running time of each BOINC application, run with and without the jailer. If there turns out to be significant difference in performance we will try to explain and research the differences.

The numbers of the benchmarks shown below are all an average of five runs to combat any possibly artifacts caused by scheduling, I/O load and other causes. During our benchmarks, the only actively running process was the jail and the benchmarked process to further combat any scheduling artifacts. Each and every application will be described in its own section and we will exemplify the numerical results. Unless stated otherwise, the program runs functionally correct with the default policy. We will not exemplify particular results if a previous section has already elaborated on the results. Therefore we recommend reading the benchmarks in order of appearance.

All benchmarks are timed with the *time(1)* program; time reports on the *real*, *user* and *sys* time.
The *real* time is the wall clock time; the time from start to finish of the program. The *user* time is the amount of CPU time spent in *user-mode*; this is only the time the process was running actively on the CPU and as such does not include time that it was blocked from the CPU by another program. The *sys* is the amount of CPU time spent in the kernel: the time spent in system calls in the kernel. We will also display the standard deviation σ to give an impression of the deviation of running time amongst individual runs.

4.5.1 Digest

The *Digest* benchmark can be ran with different hashing algorithms. The hash algorithms used is shown in parenthesis in the table after the name of the program. Here we ran the benchmark on a 288 Megabytes large file; with the

blocksize set to 8192 and have the *Digest* program repeat the hashing process 10 times.

Each value in this table is the average of five runs. Time is expressed in seconds.

Digest: $N = 5$,*Hash=md5, Blocksize=8192, Filesize=288MB*

Computational Program	Real Time (s)	User Time (s)	System Time (s)
Digest	14.49	12.33	2.13
Digest σ	1.53	2.01	0.76
Digest (jail)	91.56	50.77	46.84
Digest (jail) σ	106.52	71.57	42.42

There is a rather large difference between digest running inside the jail and running outside of the jail. This can be explained by the amount of *read* system calls required to read a large file with a relatively small blocksize.

The overhead when being run in the jailer is a system call - the process will be stopped before and after each system call while the control is being returned to the jailer to inspect the process.

These process context switches[5] take a significant amount of time. While the prisoner is suspended, the jailer also has to perform several *ptrace* system calls, at least two to:

- Find out what system call is being made. (**PTRACE_GETREGS**)
- Resume or kill the process (**PTRACE_SYSCALL**)

The jailer has to make these *ptrace* system calls twice for each system call: for the pre and post system call states, in total the very minimal amount of system calls required for one prisoner call is thus four. If the jailer has to modify the system call or its arguments, it will require several more *ptrace* system calls.

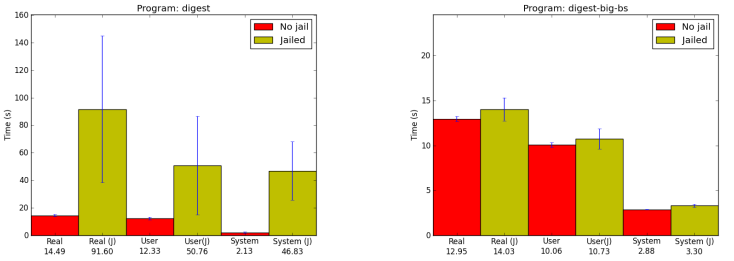
More proof of the fact that the context switches and ptrace system calls account for most of the performance overhead is presented in Section 6.3, where this overhead is eliminated for specific system calls with the help of a Linux kernel patch to achieve performance very close to running the program without the jailer.

The relatively high standard deviation of the benchmark in the jail is particularly interesting. After extensive research we discovered two things. First, running another program (in our case top(1)) while running the benchmark positively impacted the performance of the benchmark. The benchmark would run nearly **twice** as fast if a top(1) program was also running. The second observation was that when limiting the amount of CPUs in Linux to 1 [6], the *Digest* benchmark ran even faster in the jail. Running top(1) concurrently with the benchmark made no difference when only one CPU was enabled.

During some of the runs of the Digest program in the jail, top was in fact running. We have performed more benchmarks of the Digest program to investigate the huge standard deviation; and the standard deviation was minimal as long as we did not mix the "top" benchmarks with the benchmarks where top(1) was not running.

[5] Resuming a process in favour of another process. This is the basis for all multitasking operating systems

[6] with maxcpus=1 as boot argument



(a) Digest with a small blocksize (b) Digest with a large blocksize

Figure 4.1: Digest benchmark: small and large blocksizes

We also ran the digest program without the jailer; and there was no noticeable impact of top. Running the digest program with strace resembled the performance under the jailer, which means we can also rule out a strange issue with the jailer.

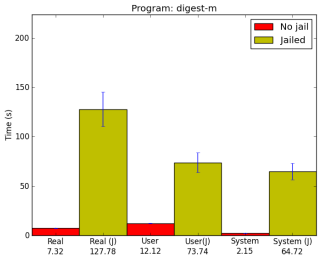
Now the main question is: why does running top concurrently impact the performance of the jailer? We believe the answer lies with the Linux kernel scheduler. The influence of top is either due to the operations it performs on the */proc* file system, or simply because it is another active task that has to be scheduled by the Linux kernel. Running a process other than top(1) or htop(1) seemed to have no effect on the performance. Switching from one CPU to another is costly and if the jailer and the digest program are constantly switching from one cpu to another, this could explain the fact that the benchmark with the jailer ran faster with just one cpu enabled. The fact that top is an active process might also help in preventing the jailer and digest program from switching from one cpu to another, as the top program is also running on a cpu. We could further eliminate any possibilities by assigning the jailer to one cpu and the prisoner to another cpu, thus avoiding processes switching from one cpu to another. We have not researched this due to time constraints.

The *Digest* program reads the file that is to be hashed in blocks of the previously mentioned blocksize (8192). This value is passed directly to the *read* system call. The larger the blocksize, the lesser the amount of *read* system calls required. In the next *Digest* benchmark, we have increased the blocksize from *8192* to *10485760* (10MB).

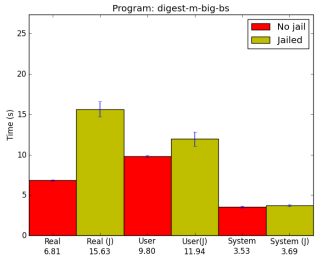
Digest: $N = 5$,*Hash=md5, Blocksize=1024*1024*10, Filesize=288MB*

Computational Program	Real Time (s)	User Time (s)	System Time (s)
Digest	12.96	10.06	2.88
Digest σ	0.56	0.52	0.05
Digest (jail)	14.04	10.74	3.30
Digest (jail) σ	2.55	2.24	0.31

As one can see, the jailer is now a lot more competitive. To illustrate the difference in the amount of system calls, we have ran the *Digest* program with the two blocksizes in another *ptrace* application that counts the amount of (read) system calls made:



(a) Digest-M with a small blocksize



(b) Digest-M with a large blocksize

Figure 4.2: Digest-M benchmark: small and large blocksizes

Calls made to the *read* system call

	Blocksize=8192	*Blocksize=1024*1024*10*
Times **read(2)** was invoked	367941	441

4.5.2 Digest-Multi

The *Digest-Multi* benchmark is mostly similar to the previously mentioned *Digest* (Section 4.5.1) benchmark. The only difference is that this program makes use of multiple cpus on a machine, where possible.

Each value in this table is the average of five runs. Time is expressed in seconds.

Digest-Multi: $N = 5$,*Hash=md5, Blocksize=8192, Filesize=288MB*

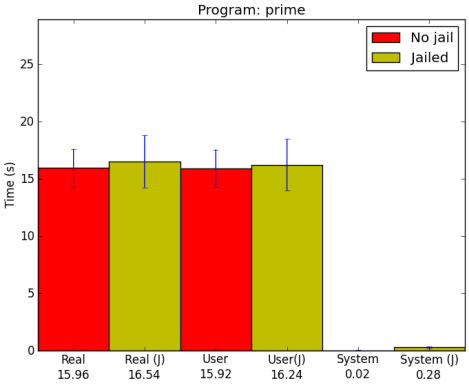
Computational Program	Real Time (s)	User Time (s)	System Time (s)
Digest-Multi	7.32	12.12	2.15
Digest-Multi σ	0.55	0.48	0.25
Digest-Multi (jail)	127.78	71.74	64.73
Digest-Multi (jail) σ	34.94	20.03	16.41

It seems that the jailed *Digest-Multi* is actually running slower than the normal jailed *Digest* benchmark.

This is explained by the fact that the jailed version had no way to determine the amount of cores available in the machine. The Python module then defaults to a number of one cores. When running the test again, but allowing access to two extra files: */proc/stat* and */proc/cpuinfo* which contain information about the amount of cpu cores/processors available in the system, the *Digest-Multi* program suddenly completed a lot faster:

Digest-Multi: $N = 5$,*Hash=md5, Blocksize=8192, Filesize=288MB*

Computational Program	Real Time (s)	User Time (s)	System Time (s)
Digest-Multi (jail)	51.2016	34.3498	32.0912
Digest-Multi (jail) σ	8.98	6.02	4.98



(a) Prime benchmark

4.5.3 Prime

In our example, we calculate the first *100000* prime numbers. Once these numbers have been calculated, we write them to a file.

Prime: $N = 5$, *100000 Prime Numbers*

Computational Program	Real Time (s)	User Time (s)	System Time (s)
Prime	15.97	15.92	0.02
Prime σ	3.35	3.31	0.04
Prime (jail)	16.54	16.24	0.28
Prime (jail) σ	4.59	4.52	0.16

The impact of the jailer was barely noticeable with the Prime benchmark. This is largely explained by the fact that the prime program hardly does any IO operations; at least until it is finished calculating the prime numbers. Once the program has finished calculating the prime numbers, it does several **write(2)** system calls to write the results to a file.

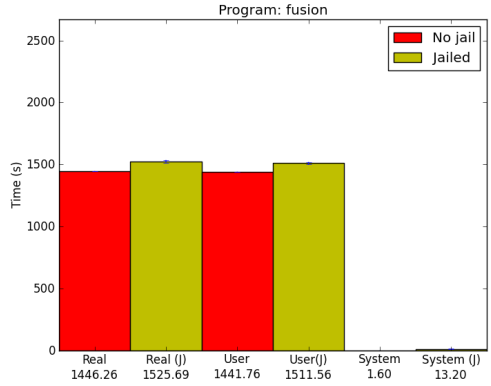
4.5.4 Fusion

The Fusion benchmark has an input file that is approximately 70 megabytes in size; it reads the file in chunks of 4096 bytes. As seen previously, we expect see a performance impact due to this when being run in the jail.

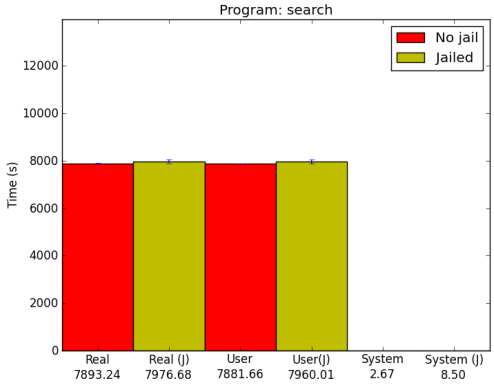
Fusion: $N = 3$

Computational Program	Real Time (s)	User Time (s)	System Time (s)
Fusion	1446.27	1441.77	1.60
Fusion σ	2.03	1.59	1.07
Fusion (jail)	1525.70	1511.57	13.20
Fusion (jail) σ	21.16	17.06	5.43

As one can see, the impact of the jailer is not all that large (5%). The differences are explained by both the IO intensive start and the amount of system calls the program does during computation, as can be seen in Appendix E. Fusion spawns a thread and checks for the resource usage of the thread group



(a) Fusion benchmark



(a) Search benchmark

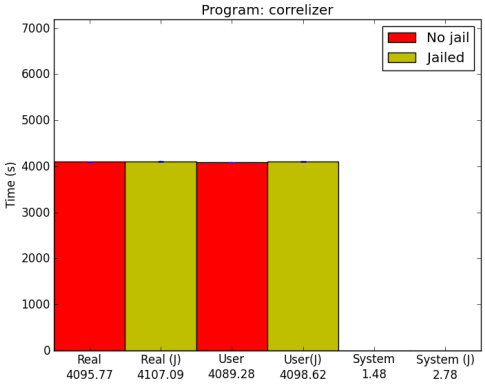
quite often, which also results in several system calls per second. In the long run, the small effect on the performance is notable.

4.5.5 Search

The Search program has a very small input file: 3.6 kilobytes. Just like the Fusion program, the Search program also checks for resource usage during the execution. However, the difference between the program in the jail and outside the jail is even less significant (1%). This is explained mainly by the fact that the Search program hardly does any IO.

Search: $N = 3$

Computational Program	Real Time (s)	User Time (s)	System Time (s)
Search	7893.25	7881.66	2.68
Search σ	25.65	25.82	0.33
Search (jail)	7976.68	7960.01	8.50
Search (jail) σ	163.79	162.91	0.69



(a) Correlizer benchmark

4.5.6 Correlizer

The Correlizer benchmark has an input file of 73 Megabytes. The blocksize it passes to the read system call is 65536. As previously shown in the Digest benchmark, we expect a blocksize of this size to have a positive impact on the performance inside the jailer compared to a smaller blocksize.

Correlizer: $N = 3$

Computational Program	Real Time (s)	User Time (s)	System Time (s)
Correlizer	4095.774	4089.28	1.48
Correlizer σ	11.44	11.51	2.21
Correlizer (jail)	4107.09	4098.63	2.78
Correlizer (jail) σ	26.32	25.26	2.71

Of all the BOINC program benchmarks, Correlizer was by far the least hindered when running in the jail. There are a few reasons for this, most notably the large blocksize that Correlizer uses for its read calls. Unlike the other two BOINC programs, Correlizer also does not constantly check the resource usage of the program. Once the program has read its data, Correlizer actually performs virtually no system calls until the calculation is finished.

4.6 Discussion

Worth noticing is the correlation between a lot of system calls in the jail and the amount of time spent in the kernel (*System Time*). In all cases, programs run in the jail have a higher system time than programs run without the jail. This should be no surprise; as the *ptrace* system call used by the jailer also counts towards the time spent in the kernel.

The *User Time* is also higher when being run in the jail; because the time the jailer spends checking policies and system arguments is also counted towards user time.

The benchmarks are not clear on where exactly the system calls take place. However, all the IO system calls are done within a short period of time, we can roughly divide the grid prisoners into two states: the first state, where the prisoners perform lots of IO system calls and the second state where they only perform computations and hardly use any system calls.

We have not researched the possibility of limited the resources of the prisoner, by only allowing a certain amount of system calls per second for example. If resource restrictions are put in place, it may be a good idea to allow the prisoner to "burst" (perform more system calls per second than usual) for a short while - this may help increasing the performance while the prisoner is doing a lot of IO system calls.

Changing the default (provided) policy has not been necessary in our test cases. Most of the applications in the grid are even either statically linked or ship with their own libraries rather than using the system libraries. (To ensure the availability of the right versions of the libraries required)

It is remarkable that all the tested BOINC applications work with such a small and strict policy. This is at least partially thanks to glibc, which picks sane defaults for applications in case of missing configuration files; as well as handle errors in case specific functionality is not available because it has been restricted by the jailer.

The default policy should suffice for all trivial public resource grid applications. As of yet, all TCP and UDP traffic is forbidden by the jailer; this should however not affect Desktop Grid applications as they should have no need for TCP and UDP traffic[7].

As stated before, most programs work with the default policy; but the policy exposes very little information to the programs. In some cases we imagine it may be useful to allow access to a few more files.

For example, the files */proc/cpuinfo* and */proc/stat* contain some information about the hardware and kernel, these in no way help the prisoner to escape the jail but can provide the prisoner with hints for its execution; Python[p:p] uses both these files to determine how many cpus are available in the *multiprocessing* module; if the module cannot access these files; the default value is 1. This can seriously cripple performance on multicore systems as only one core will be used by the grid application. */proc/meminfo* might also be of some use.

Another useful file which would be very useful to put in the default policy is the */etc/ld.so.cache* file, which contains a compiled list of candidate libraries found in the system library path.

Other files such as */etc/inputrc* and the directory */etc/terminfo* are quite useful to have a more functional terminal. This does not really matter as most grid programs do not use virtual terminals; but when running interactive prisoners in the terminal it is useful if the prisoner can access these files.

Java[jav] requires some policy changes. A major problem with Java is the absence of a default installation path. The way Java is installed seems to vary

[7] Communication in grids is quite common; but it is not viable in public resource computing.

greatly per GNU/Linux distribution; and each GNU/Linux distribution provides its own methods to switch between Java versions.

For example, to find out which Java version should be launched, the GNU/Linux Debian distribution needs to read the file */etc/alternatives/java*, whereas Gentoo Linux reads */etc/java-config-2/current-system-vm*. The problem lies in the fact that both these files are stored in */etc*, to which the default policy allows no reading or writing unless explicitly granted, as with the */etc/localtime* file.

Another problem with Java is the fact that Java relies [8] on the special */proc/self/exe* file to locate and load libraries. The */proc/self/exe* file contains a link to the current executable. Access to */proc* is disallowed unless explicitly granted. When enabling read access to this file, Java runs fine.

Access to */proc/self/exe* should generally be safe, except for programs contained in the jailer virtual directory - by reading the */proc/self/exe* link the prisoner may be able to find out the true directory the prisoner resides in. Wether this a true security issue is debatable, but this is definitely a privacy issue.

As it turns out, running Java in the jailer does require some changes in the policy.

[8] http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6189256

Chapter 5

Related work

Modern computers have sufficient resources to make *virtualisation* - the creation of a virtual hardware platform or virtual environment in which applications are ran (in an isolated environment) - viable. Virtualisation is not limited to just a hardware platform, network or storage virtualisation should be familiar as well, think about virtual private networks and local area networks.

Full virtualisation is an alternative way of separating a process from the rest of the system, by starting an entire (virtualised) operating system and running the process in this virtualised operating system. A downside is that virtualising an entire operating system is complex and quite resource intensive because an entire operating system is run just to jail a specific application.

Full virtualisation is not a form of jailing; as it does not trace and intercept specific actions of a process as a jailer would. Jailing however, can be a form of virtualisation: rewriting specific calls to the network or file system is basically just virtualising the network and file system.

Virtualisation enjoys other advantages not related to confining processes such as live migration, but it is not relevant to this research.

Presented below is an overview of related work, in particular the Condor High Throughput System, three virtualisation methods for Linux, two alternative jailing implementations and one *access control* system.

5.1 Condor

The **Condor High Throughput System** [LLM88] is an open source framework for high throughput computing aimed at distributed parallelization of computationally intensive programs. Originally presented in 1988, it is currently maintained by the Condor team at the University of Wisconsin-Madison.

The Condor framework guarantees that all jobs will eventually complete, doing as little duplicate work as possible as possible; in contrast to BOINC which has a feature to perform all computations at least N amount of times to prevent falsification of results due to malicious intents or system failures.

Condor achieves requiring as little work as possible with so called *Checkpointing* [LLM88, con].

> Checkpointing is the saving of the state of a program during its execution so it can be restarted at any time, on any machine in the

system.

This interesting feature is achieved by linking the program with the Condor system call library. (libcondorsyscall.a)

Another way to achieve the same effect as this so called checkpointing would be to use the Cryopid program, which allows saving the states of processes to resume them later on, even after a reboot or shutdown of the system. [cry] and [che]

5.2 KVM

KVM, the Linux Virtual Machine Monitor [KKL+07], is a Linux subsystem that uses the currently provided x86 virtualisation instructions by Intel and AMD to allow the creation and execution of multiple virtual machines in Linux.

5.2.1 Guest mode

These virtualisation instructions require the introduction of an extra operating mode: the *Guest mode*. Currently, Linux has two operating modes; the *Kernel Mode* and the *User Mode*. In Kernel Mode, every instruction in the x86 instruction set can be executed, this in contrast to user space mode in which not all instructions can be executed; this is implemented within the x86 instruction set.

The processor, if it supports the virtualisation instructions, can switch to a new guest operating mode which has the same rights as user mode, with the exception that certain instructions or registers can be trapped by system software.

5.2.2 Integration and Security

As previously described, KVM does full hardware virtualisation. Emulation (and Virtualisation) projects like QEMU [Bel05] are able to take advantages of KVM to tremendously speed up their performance. Real emulation is very slow while KVM allows near-hardware speed.

5.3 Xen

Xen[BDF+03] is an x86 virtual machine monitor which allows sharing conventional hardware without sacrificing performance or functionality on several UNIX host operating systems.

5.3.1 Paravirtualisation

In contrast to KVM, Xen provides *Paravirtualisation*. Paravirtualisation provides a software interface to the virtual machines that is not identical to the real (underlying) hardware. The advantage of paravirtualisation is that it allows certain operations that perform worse in a virtualised environment to run in a non-virtualised environment by providing specially defined "hooks" to which the virtualised guests can requests tasks. This can lead to a more simplistic

virtual machine manager by relocating the execution of critical tasks to the host domain, as well as reduce the overall performance hit of the machine code execution in the guest.

The downside of this approach is that the guest operating systems needs to be modified using special hypercall ABI. Guest *applications* however, typically need not be modified.

5.3.2 Intel and AMD virtualisation instructions

Xen, as of version 3.0, also supports x86 (hardware-assisted) virtualisation using the virtualisation extensions provided by Intel and AMD, which allows it to run certain unmodified versions of Windows as well as versions of Linux.

5.4 Janus

Janus is a jailing application originally published in 1996. The project has not seen any recent development; the latest version of Janus is an alpha version for Linux 2.2. [1]

Janus shared a design goal with the jailer as presented in [Zup10] - Janus was also written with security through simplicity in mind. The entire codebase of Janus at the time of publishing was 2100 lines of code.

Janus' other main design goals are:

- Security: The prisoner should not be able to escape the jail and harm or compromise the underlying system. - Versatility: Individual system calls should be allowed or denied in a flexible way - depending on the arguments. - Configurability: Different programs require different policies.

To trace applications, Janus does not use ptrace. Instead Janus uses the **/proc** interface provided by **Solaris 2.4**. [2] , in favour of ptrace, **man(1)** points out a few flaws of ptrace:

> This page documents the way the ptrace() call works currently in Linux. Its behaviour differs noticeably on other flavours of UNIX. In any case, use of ptrace() is highly OS- and architecture-specific.
>
> The SunOS man page describes ptrace() as "unique and arcane", which it is. The proc-based debugging interface present in Solaris 2 implements a superset of ptrace() functionality in a more powerful and uniform way.

It is for these reasons that Janus was written for Solaris 2.4 and not for other platforms that supported ptrace. [GWTB96] mentions the inability of ptrace to only trace specific system calls instead of all system calls as a reason to use the **/proc** interface in favour of ptrace. This is apparent in the Janus optimiser in which they note that the system call **write(2)** is always allowed. Not having to trace this system call at all reduces the overhead considerably considering **write(2)** is a common system call.

We experimented with a new option to ptrace that would allow for similar behaviour in Section 6.3.

[1] http://www.cs.berkeley.edu/~daw/janus/releases.html
[2] https://secure.wikimedia.org/wikipedia/en/wiki/Solaris_(operating_system)

5.5 BSD Jail

The FreeBSD jail, as described in [KW00] is a way to virtualise processes on a FreeBSD system, by creating minimal independent "systems" called *jails*. It is implemented on the operating system layer. All the processes in a *jail* can only access resources enclosed within the jail, each jail has support for a *root* user, but even the *root* user in a jail will find its permissions do not extend beyond the jail. Each jail has its own virtual filesystem and networking stack, including its own IP address. Processes can share a *jail*, but once they enter the jail there is no way to leave the jail.

One particular advantage of the BSD jail is that it does not impose a lot of policy management, unlike systems like SELinux (Section 5.7). A downside of having a lot of system policies is that it places a huge burden on the system administrator. It is hard to ensure that the policies are really secure and the system administrator may too quickly believe that they are in fact secure, another issue is that the administrator will simply get too frustrated and hardly enforce any policies.

5.6 Linux VServer

5.6.1 Background

The Linux-VServer[vse] concept differs from KVM and Xen in the sense that it separates the user-space environment in such a way that it looks like a real environment to the processes contained within, even though the processes do not run in a fully virtualised environment as is the case with full virtualisation. A huge advantage of this approach is that the overhead of loading an extra operating system is gone.

The Linux Kernel already provides many security features that Linux-VServer utilises to create its secure environment. Features like the Linux Capability System, Resource Limits, File Attributes and Change Root environment are used.

5.6.2 Confinement method

One major difference between the approaches previously seen at KVM and Xen (Virtual Machines) is that VServer does not have the virtualisation part as a "side-effect"; you only virtualise when required.

The Linux-VServer does require (Linux) kernel modifications in order to create and maintain a secure environment. Amongst changes to the kernel are:

- Context Separation, to hide all processes outside the "context" scope and prohibit any unwanted interaction between processing within different contexts.
- Network Separation, special addresses like *IPADDR_ANY* or the *localhost* address have to be handled in a special way.
- Resource Isolation, lots of resources may be shared amongst different contexts. Amongst these are: Shared Memory, IPC, User and process IDs, Unix pyts and sockets.

5.7 SELinux

Security-Enhanced Linux is a feature in the Linux kernel that implements a security mechanism for supporting access control policies. [LS01] Developed by the NSA in 2000, it was integrated in the Linux kernel in 2003. In particular, it brings *Mandatory Access Control* security by taking advantage of the Linux Security Modules, a framework for implementing security modules.

Mandatory Access Control is a means a restriction the access of a specific object on some subject. An object is typically a process and a subject a file, shared memory segment or internet port. Access from an object to a subject will be run against a set of authorisation rules[3] to determine whether the access can be allowed. A user can not change this policy; this in contrast to *Discretionary Access Control*. With DAC, objects typically have an *owner* who controls the permissions of said object.

DAC is used in Unix for file modes. MAC allows more fine granted security than DAC, DAC on itself is not sufficient to protect against malicious code because every program run by a user has all the privileges of that user and thus have access to all the objects owned by the user.

[3] policy

Chapter 6

Future work

6.1 Threaded jailing

When benchmarking applications inside the jail, we found that the jailer was fast enough to immediately handle of the system calls made by the prisoners.

However, we expect that once the jailer has several prisoners which are all performing system call intensive tasks, it is possible that the jailer cannot handle all the system calls at right away and effectively keeping the prisoners waiting for their system calls to be denied. We believe that using the *Digest-Multi* benchmark on a system with enough cores will be eventually limited by the jailer.

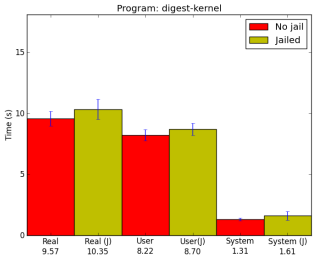
A multithreaded jailer should theoretically be able to cope with several prisoners performing system call intensive tasks. Each thread could perform *wait(2)* on all of the children, effectively parallelising balancing the load of all the prisoners over several jailer threads.

6.2 GPU Policies

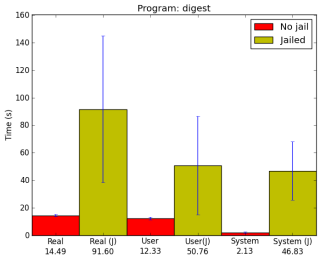
With the upcoming demand for offloading calculations to Graphics Processing Units (GPU), an interesting point of research would be how the policy could be altered to run OpenGL and more importantly OpenCL applications.

Quick research showed that the only access required for a hardware accelerated 3D program in Mesa[mes], is a special device representing the card in */dev/dri/cardN*. Mesa talks to the card by using ioctl calls.

Aside from the access required to talk to the graphics card driver, it is also important to consider other possible security risks: What can a program do with access to the graphics card driver? Can the program talk to other programs (IPC) using the graphics drivers, can it modify other programs or read their memory? Can we prevent this with the jailer? Is it possible to cause a denial of service with access to the graphics card? Can we prevent this with resource management?



(a) Digest benchmark with ptrace kernel patch



(b) Digest benchmark without ptrace kernel patch

6.3 ptrace changes in Linux

As the *Performance* section shows, the performance of a program in the jail is impacted mainly by the extra context switches and system calls required to allow a system call to continue. In most cases, this should not be necessary simply because there are currently *120* system calls always allowed by the jailer, including *read(2)* and *write(2)*.

We believe that it should be possible to extend the ptrace API with an option to always (or never) report certain system calls, allowing calls like *read* and *write* to execute unhindered by the jailer. We expect that this will provide the jailer and its prisoners with an tremendous performance boost.[1] This has also been shown in [tNBH+07].

A simple performance benchmark can be seen in 6.1a, where the worst performing program in our benchmarks (Digest) is run without the jailer and with the jailer that uses the kernel patch, to ensure that the Digest benchmark is not stopped on **read(2)** and **write(2)** calls. The Digest benchmark uses the default (small) blocksize. With the kernel patch, the performance increases significantly. [2]

6.4 Mixed ABI in the jailer

While the jailer should mostly be secure, a few known issues remain. A big issue is the fact that 64 bit applications can perform 32 bit system calls. When this happens, ptrace will stop the prisoner as usual and wait for the jailer to do its work. However, the jailer cannot (easily) find out if the prisoner is doing a 64 bit or 32 bit system call. Since the numbers of the system calls differ per ABI, it is not possible to determine the system call that is being executed by the prisoner without first knowing the mode it is running in.

A solution to this problem would be support from the Linux kernel; a way to get the mode of the current system call, the best way would be to deliver this in-

[1] We have in fact written such as kernel patch: the performance of programs such as Digest were near native (that is, when not being traced) when always allowing the *read* and *write* system calls.

[2] There is a difference in time between the native run in the two figures; this is because they operated on a file a different size. The kernel-patch benchmark was run at a much later time and the original file used for benchmarking has been lost. However, the comparison with and without the patch does not suffer much from this small limitation.

formation in the system call trap signal when using the *PTRACE_O_TRACESYSGOOD* option. However, this requires kernel changes and is therefore not an immediate solution.

Another solution would be to analyse the assembly instruction that was used to perform the system call. On x86 a system call is performed with the *int 0x80* instruction whereas a x86-64 instruction is performed with the *syscall* instruction. To prevent a race condition similar to the one presented and solved in [tNBH+07], the memory pages should be either writable, or readable and executable. This way we can assure that the prisoner does not quickly change the instruction in another thread - before we read but after the initial system call has been made. Another advantage of this approach it is possible to "cache" the system call instruction. Because it is not possible to change the instruction without changing the permissions of the memory. It is possible to temporarily give write access to the memory and then update the system call instruction cache.

6.5 Jailer test suite

While we have previously shown that the jailer performance is more than adequate, we have not shown that the jailer is secure: by design the jailer should be secure, but bugs could exist in the source code of the jailer. To effectively combat regressions and other bugs, we believe a test suite of some sort would greatly increase confidence that the jailer is in fact secure. Another interesting idea would be to run actual malware in the jailer and examine how well the jailer shields the computer from malware.

Chapter 7

Discussion

Integrating the jailer in BOINC and other grid environments would be a significant advancement in security for grid clients. As shown by the micro benchmarks, the performance penalty of the jailer is not really noticeable, especially since most grid applications do not use a lot of system calls. Using the built-in resource manager of the jailer would be a useful addition to prevent denial of service attacks; limiting the amount of cpu-time the system calls of the prisoner can use[1].

If a specific application requires modifications to the default policy the grid project could opt to ship an additional (small) policy that would be used by the jailer as an addition to the default policy. This approach is sub-optimal, but auditing small policy files is a lot easier than auditing large code bases.

Shipping policies with grid programs can still be unsafe in cases where a third party may tinker with the policies being distributed. Great care should be taken in the distribution of additional policies, signing and verifying the integrity of extra policy files is therefore highly recommended.

What the jailer does not provide is privacy; the goal of the jailer is to create a secure environment that a prisoner cannot escape from.

The jailer does not really enhance privacy; it simply changes the user name and a very minimal */etc/passwd* file; but the jailer does not protect the system from a prisoner collecting information about the system (such as installed programs) in directories that the prisoner can access; such as */usr*. Typically the directories currently allowed hold no personal content - personal content is typically stored in */home* or */etc*.

For example, it is to some extent possible to detect the GNU/Linux distribution the prisoner is running on: Gentoo Linux stores their Package tree in */usr/portage*; if this directory is present it is reasonable to assume that the prisoner is running on Gentoo Linux. The prisoner could also examine the installed packages and look for the package manager frontend; such as *emerge*.

Privacy protection is a different issue than protecting and securing the operating system from malicious programs and has not been a concern in this

[1] It would be particularly useful to allow "bursting" of cpu-time for the system calls; especially because most of the system calls of the prisoner are made at the start of the program

research.

Compared to other alternatives that can be used for jailing applications such as *KVM*, *Xen* the jailer is a much more lightweight alternative as it does not require loading and virtualising an entire operating system as well as kernel changes or administrator permissions. While the goal of this thesis was not a comparison of the jailer, *KVM*, *Xen*, *VServer* and *BSD Jail* it would certainly be interesting to analyse and compare the performance of each virtualisation or jailing mechanism. The overhead of creating an entire operating system is also something the jailer elegantly works around by only virtualising (rewriting) the paths where required, whereas with Linux VServer the administrator creates an entire system (albeit with a shared kernel) and with mechanisms like KVM even an entire kernel is ran. For isolating one (group) of processes, simply jailing the processes seems like a more elegant solution.

However, there are drawbacks to this ptrace jailing approach as well. If the jailer contains a bug that allow the prisoners to escape, the prisoners might be able to do considerable more damage to the main system. While this is also possible in a completely virtualised operating system, it is generally harder to break out of the virtualised operating system. Full virtualisation like *KVM* does also comes with certain other advantages - like the easy of moving an entire operating system and all its processes to another machine, but these advantages are not relevant to jailing processes on desktop grids.

At this point, the jailer is functional but not completely secure (Section 6.4). The chances of a 64 bit program performing a 32 bit system call is slim, unless the program would be written just to circumvent the jailing mechanism. However, we plan to fix this security issue in the near future.

We don't believe threaded jailing is a must-have for running applications in the jail; having a threaded jailer would only become interesting when the jailer is used to manage a lot of different tasks. Unless these tasks have to communicate, it should be possible to simply start each task with its own jailer.

However, speeding up ptrace would be a great addition.[2] The patch we created as a proof of concept however is not ready to be included in the Linux kernel. An alternative way to implement handling only specific system calls is also possible with a relatively new feature called "Secure Computing" (SECCOMP) which is available as of Linux 3.5. [3]

[2]Speeding up ptrace is a different optimisation from threaded jailing.

[3]http://kernelnewbies.org/Linux_3.5#head-c48d6a7a26b6aae95139358285eee012d6212b9e

Bibliography

[ACK+02] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45:56–61, November 2002.

[AGZ07] Csaba Marosi Attila, Gombs Gbor, and Balaton Zoltn. Secure application deployment in the hierarchical local desktop grid. 2007.

[AKS98] Albert Alexandrov, Paul Kmiec, and Klaus Schauser. Consh: Confined execution environment for internet computations. 1998.

[And04] David P. Anderson. Boinc: A system for public-resource computing and storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, GRID '04, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[Bel05] Fabrice Bellard. Qemu, a fast and portable dynamic translator. 2005.

[che] Checkpointing. http://checkpointing.org.

[con] Condor checkpoints. http://www.cs.wisc.edu/condor/manual/v6.8/4_2Condor_s_Checkpoint.html.

[cor] Correlizer website. http://svahesrv2.bioquant.uni-heidelberg.de/correlizer/.

[cry] Cryopid. http://cryopid.berlios.de.

[edg] Edges code signing. Personal communication with someone from the EDGeS team.

[Fer10] Dario Ferrer. Isdep, a fusion application deployed on a volunteer computing platform. 2010.

[FGNC00] Gilles Fedak, Ccile Germain, Vincent Nri, and Franck Cappello. Xtremweb : A generic global computing system. 2000.

[gdb] Gnu debugger website. https://www.gnu.org/software/gdb/.

[GWTB96] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. 1996.

[jav] Java programming language website. http://java.com.

[KKL+07] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguor. kvm: the linux virtual machine monitor. 2007.

[KLM+08] Gabor Kecskemeti, Oleg Lodygensky, Attila Marosi, Zoltan Balaton, Gabriel Caillat, Gabor Gombas, Adam Kornafeld, Jozsef Kovacs, Haiwu He, and Robert Lovas. Edges: Bridging egee to boinc and xtremweb. 2008.

[KW00] Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. 2000.

[lin] Linux standard base. http://en.wikipedia.org/wiki/Linux_Standard_Base.

[LLM88] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[LS01] Peter A. Loscocco and Stephen D. Smalley. Meeting critical security objectives with security-enhanced linux. 2001.

[man] See chapter 5 of the manual pages; section "proc": man 5 proc.

[mes] Mesa 3d. http://mesa3d.org/.

[MGB+07] Attila Csaba Marosi, Gbor Gombs, Zoltn Balaton, Pter Kacsuk, and Tamas Kiss. Sztaki desktop grid: Building a scalable, secure platform for desktop grid computing. 2007.

[NOT+] GJ Vant Noordende, BJ Overeinder, RJ Timmer, FMT Brazier, and AS Tanenbaum. A common base for building secure mobile agent middleware systems. In *Proc. Workshop on Agent Based Computing IV (ABC07), 2007, pp. 13-25.*

[OLW97] John K. Ousterhout, Jacob Y. Levy, and Brent B. Welch. The safe-tcl security model. pages 271–282, 1997.

[p:p] Python programming language website. http://python.org.

[RBM+] Alejandro Rivero, Andres Bustos, Attila Marosi, Dario Ferrer, and Fermin Serrano. Isdep, a fusion application deployed in the edges project.

[tNBH+07] Guido Van 't Noordende, Adam Balogh, Rutger Hofman, Frances M. T. Brazier, and Andrew S. Tanenbaum. A secure jailing system for confining untrusted applications. In *International Conference on Security and Cryptography (SECRYPT)*, 2007.

[vNBTa] G van't Noordende, FMT Brazier, and AS Tanenbaum. Guarding security sensitive content using confined mobile agents. In *Proc. 22nd ACM Symposium on applied Computing (SAC), Seoul, Korea, march 2007, pp. 48-55.*

[vNBTb] GJ van't Noordende, FMT Brazier, and AS Tanenbaum. Security in a mobile agent system. In *1st IEEE Symposium on Multi-Agent Security and Survivability (MASS), Philadelphia, USA, 2004, pp. 35-45.*

[vse] http://linux-vserver.org/Paper.

[Zup10] Indan Zupančič. Redesigning a linux jailing system. 2010.

Appendix A

Extracting program arguments from BOINC

```
argv[0] = buf;
parse_command_line(cmdline, argv+1);

/* Open the file in appending mode */
FILE *f = fopen("/tmp/arguments.out", "a+");
char **argi = argv;
int fooi = 0;

fprintf(f, "New Process\n");

/* Walk over the argument list */
while (**argi) {
    fprintf(f, "\t%d: %s\n", fooi++, *argi);
    argi++;
}

/* Close the file */
fclose(f);

retval = execv(buf, argv);
```

Appendix B

Digest in Python

```
import time, hashlib

import sys

_file = sys.argv[1]
_meth = sys.argv[2]
_amt = int(sys.argv[3])
_bs = int(sys.argv[4])

t = time.time()

for x in xrange(_amt):
    m = getattr(hashlib, _meth)()
    f = open(_file)
    for l in iter(lambda: f.read(_bs), b''):
        m.update(l)

    s = m.hexdigest()

print time.time() - t
print s
```

Appendix C

Multiprocess Digest in Python

```
import time, hashlib

import sys

import multiprocessing

_file = sys.argv[1]
_meth = sys.argv[2]
_amt = int(sys.argv[3])

cpu = multiprocessing.cpu_count()

t = time.time()

def f():
    m = getattr(hashlib, _meth)()
    f = open(_file)
    for l in iter(lambda: f.read(8192), b''):
        m.update(l)

    s = m.hexdigest()

for x in xrange(0, _amt, cpu):
    p = []
    for y in xrange(cpu):
        p.append(multiprocessing.Process(target=f, args=[]))

    for x in p:
        x.start()

    for x in p:
        x.join()
```

```
    del p

print time.time() - t
```

Appendix D

Prime number generator in Python

```
#!/usr/bin/env python

import sys
amount_to_compute = int(sys.argv[1])

def compute_prime_numbers(to):
    """
    Finds the prime numbers less than *to* and returns them as a list.
    """
    prime = []

    n = 2
    while n <= to:
        n = n + 1
        is_prime = True
        for p in prime:
            if n % p == 0:
                is_prime = False
                break

        if is_prime:
            prime.append(n)

    return prime

primes = compute_prime_numbers(amount_to_compute)


import json

f = open('out.txt', 'w+')
f.write(json.dumps(primes))
```

```
f.close()
```

Appendix E

Fusion System call breakdown

```
Syscall arch_prctl called 1 times.
Syscall brk called 5 times.
Syscall clone called 1 times.
Syscall close called 58 times.
Syscall execve called 1 times.
Syscall fcntl called 57 times.
Syscall fstat called 56 times.
Syscall getrlimit called 1 times.
Syscall getrusage called 15447 times.
Syscall lseek called 1 times.
Syscall lstat called 31 times.
Syscall mmap called 55 times.
Syscall mprotect called 1 times.
Syscall munmap called 53 times.
Syscall nanosleep called 15418 times.
Syscall open called 59 times.
Syscall read called 18826 times.
Syscall rt_sigaction called 18 times.
Syscall rt_sigprocmask called 24 times.
Syscall set_tid_address called 1 times.
Syscall setitimer called 1 times.
Syscall stat called 58 times.
Syscall uname called 1 times.
Syscall unlink called 1 times.
Syscall write called 37 times.
```

Appendix F

Search System call breakdown

```
Syscall _llseek called 7 times.
Syscall access called 1 times.
Syscall brk called 3 times.
Syscall clone called 1 times.
Syscall close called 101 times.
Syscall dup3 called 2 times.
Syscall execve called 1 times.
Syscall fcntl64 called 2 times.
Syscall fstat64 called 103 times.
Syscall futex called 2 times.
Syscall getcwd called 1 times.
Syscall getdents called 2 times.
Syscall getdents64 called 2 times.
Syscall getrusage called 8090 times.
Syscall ipc called 1 times.
Syscall mmap2 called 107 times.
Syscall mprotect called 5 times.
Syscall munmap called 92 times.
Syscall nanosleep called 8043 times.
Syscall open called 102 times.
Syscall read called 17 times.
Syscall rt_sigaction called 8059 times.
Syscall rt_sigprocmask called 16090 times.
Syscall set_robust_list called 2 times.
Syscall set_thread_area called 1 times.
Syscall set_tid_address called 1 times.
Syscall setitimer called 1 times.
Syscall stat64 called 13 times.
Syscall time called 1 times.
Syscall ugetrlimit called 1 times.
Syscall uname called 1 times.
Syscall unlink called 44 times.
```

```
Syscall write called 117 times.
```

Appendix G

Correlizer System call breakdown

```
Syscall arch_prctl called 1 times.
Syscall brk called 9 times.
Syscall close called 2 times.
Syscall execve called 1 times.
Syscall fstat called 3 times.
Syscall lseek called 1 times.
Syscall mmap called 4 times.
Syscall munmap called 3 times.
Syscall open called 3 times.
Syscall read called 2308 times.
Syscall uname called 1 times.
Syscall write called 9 times.
```