Performance evaluation of Sandboxing techniques for Peer-to-Peer Computing

G. Bosilca, F. Cappello, A. Djilali, G. Fedak, T. Herault and F. Magniette
LRI - CNRS and Paris-Sud University

Abstract

Computing in large scale distributed systems is becoming more and more attractive due to huge available computing power connected to Internet and the very low cost of their utilization. Peer-to-Peer Computing harnesses idle computing power furnished by volunteers into one unified computing resource accessible by each participant. The consequence is executions on remote resources of mobile codes provided by the users. Even strong authentication techniques don’t preclude a correctly identified user to run aggressive codes and corrupt a resource. So beside user authentication, resource security must rely on some self-defense mechanisms. Sandboxing implements self-defense by confining the execution of a code inside an unbreakable envelop. There are several sandboxing approaches: 1) byte code execution on a virtual machine 2) native code execution on a secured operating system. In this article, we compare the performance and security merits of these two approaches. We analyze 4 Java virtual machines and 3 sandboxing environments for native code execution. Unsurprisingly, the sandboxed execution of native codes outperforms Java byte code execution for all numerical benchmarks. More interestingly, Java provides similar performance than most of the sandboxing environments for I/O benchmarks. The security comparison provides less distinctive results. While using different strategies, the two approaches can provide similar security level according to same criterion. The XtremWeb Peer-to-Peer platform have been modified to integrate the Subterfugue sandbox and a specialized secured class loader for Java. Confronted with a parallelized EP from the NAS benchmark for this peer-to-peer system, executed in both native and byte-code form, the overhead of sandboxing seems negligible.

1 Introduction

Security is a first concern in High Performance Distributed Computing. Grid Computing[11] facilitates large-scale access and sharing of the computing resources of institutions. Thus Grid computing toolkits like Globus relies on security components as the basic bricks for building Grid architecture. Traditional GRID infrastructure connects trusted systems installed in High Performance Computing centers.

New emerging trends in distributed computing like Peer-to-Peer Computing and Global Computing push the security problem to another direction. Global Computing as reflected by the SETI@Home project[3] is now generally accepted as a powerful and cheap solution to achieve huge computing power. One of the most successful Peer-to-Peer systems is Napster[16]. Its basic functionality allows every user in the system to act as a client and/or as a server of the system. Most of the current Peer-to-Peer systems are related to file exchanges between users. Whatever distributed system design they are using (centralized/hierarchical or decentralized/self organizing), peer-to-peer systems basically provide the same interaction model between resources than Napster. Peer-to-peer Computing is, in principle, a merge of Global Computing and Peer-to-Peer interaction model. It enhances these models by allowing every node to provide and use the computing power of the other participants. Any node in the network may submit applications and data to any other node of the network.

Because Peer-to-Peer Computing implies the codes execution on remote resources, it gives the opportunity for aggressive codes to corrupt participant resources. Many attack techniques like distributed denial of service or virii based on buffer overflow attack are publicly described and most people can use them without a special knowledge. As witnessed by many security alerts, more and more commonly used applications are subject to various exploits and thus generally weaken the overall security of a system; to the point that other security systems have to be used beyond the basic protection provided by operating systems.

Most of the security mechanisms of the GRID systems rely on preexisting security mechanisms of the HPC centers such as the traditional login procedure which could be improved in scalability and strength by recent scalable authentication procedures such as the single sing-on used in GSI [8]. In such kind of GRID systems, the user is well identified before accessing the resources and the user trusts the resources because they belong to well identified institutions.
Peer-to-Peer computing is characterized by a huge resource volatility, an extreme code mobility, a very large number of participants (usually x 10000) and a high potential of attacks. It is very unlikely that all participants (acting as a client and/or as a server) will setup and maintain a very large authentication database. Another possible solution is to use authority certificate delivered by a trustee. However, the potential of attack launched by authenticated resources may remain high because even strong authentication mechanism could not preclude a correctly identified user to run aggressive codes and corrupt a resource. We strongly believe that in such peer-to-peer systems, beside authentication mechanisms, the resource should use a self-defense mechanism during the runtime execution, which confines the code execution inside an unbreakable envelop.

Sandboxing is a technique which implements this principle to protect the host from an hostile code. According to a security policy Sandboxing neutralizes hostile behaviors, enforces resource usage and prevents any attempt to exploit security holes on the host. Because it offers a complementary security mechanism providing a runtime control of the execution, sandboxing may appear as the cornerstone technology to allow a wide and safe use of peer-to-peer systems.

The most widely sandboxing system used in distributed systems is the Java Virtual Machine which was designed as a client-side from the beginning for safe mobile code. In particular it is possible since five years ago to download a code called “applet” from a web browser and run it on a computer in a safe way. The drawback of the Java virtual machine is the use of byte-code interpreter machine with the performance loss induced. Sun Hotspot JVM, JIT compiler and IBM selective JIT provide substantial performance improvements compared to the original Sun JVM. There is currently a large effort to improve the performance of Java with the aim to provide performance close to the one of native code execution. The Microsoft .Net[14] project also relies on an intermediate language (Microsoft IL) executed by the Microsoft Common Language Runtime. Performance are claimed to be close to one of native code because the typing rules determine well-typed IL instruction sequences that can be assembled and executed.

Since this time, several sandboxing systems have appeared to allow the execution of native code. This sandboxing techniques were primarily designed to host application known to have periodic security holes or subject to remote attack like web server. Native code execution exhibits appealing features compared to byte code execution.

It allows the execution of an application without any change on homogeneous platforms. This is likely the case for a Global and Peer-to-Peer computing systems using few different kind of resources. For example, SETI@home and some other Global Computing platforms provide their applications as binary codes for less than ten different kinds of resources that represents almost all potential participant resources connected to the Internet. Because any application can be launch on the system without requiring the source code, this open the opportunity for software companies to allow the running of their proprietary applications on a peer-to-peer non-trusted environment.

In this paper we examine two issues concerning the sandboxing techniques: 1) we present a quantitative performance evaluation of various sandboxing techniques and 2) we discuss qualitatively the security merit of each technique. The paper is constructed as following: the first section presents an overview of various Sandboxing techniques. Section 2 provides a description of the Sandboxing systems compared in this paper. The third section compares their performance for several benchmarks: lmbench for I/O operation, Scimark, Linpack and two kernels of the NAS benchmark 2.3 for numerical computations, gzip, gunzip and gcc. Section 4 discuss the performance and security merit of all Sandboxing techniques. Section 5 presents an evaluation of two sandbox solutions in the context of a peer-to-peer system. We conclude and present the perspective of this work in section 6.

2 Sandboxing Systems Investigated

Sandboxing is a mechanism which confines the execution of an application in a safe environment: the sandbox. The main goal of a sandbox is to restrain the rights that a program can have in the system down to the desired security level.

During the execution of the program, the sandboxing application performs some supplementary controls to grant or forbid access to resources like files or network access, limit the resource usage like CPU time utilization or consumption of memory, disk space or network bandwidth, and finally to detect hostile behavior.

In this section we overview different known approaches to implement a controlled execution environment.

2.1 Java Virtual Machine

Providing a secure execution environment through a virtual machine is certainly the best known and largely accepted method, mainly due to the success the Java language has gained this past years. The security architecture is deeply described in [13], we only summarize the key concepts.

The Java security model is based both on compile-time and run-time checks. Java language is strongly typed and at run-time a bytecode verifier guarantees the type safety. Other mechanisms perform array bound checking and control the origin of the code according to its signature.

The sandbox model of the Java Virtual Machine is based on the Protection domain concept. Protection domain is an
indirection between the code and the permissions granted. First, the identification of the mobile code is established by checking its provenance via URL and its signature via public keys. Then a security manager, according to a security policy, assigns permissions to the different protection domain.

[5] provides a comparison between Java, C and Fortran for numerical benchmarks. We are extending their results with the EP and CG benchmarks issued from the NAS NPB-2.3 benchmark suits and with more recent compilers. [13] also evaluates the performance of the implementation of the protection domain model.

2.2 Presentation of Sandboxing techniques for Native Code Execution

Sandboxing of native code execution relies on an interposition principle between the process and the operating system. Interposition gives the control of the execution to an inspector at any time the process performs critical operations, which allows to implement a security policy.

Interposition code can be inserted inside the native code itself. Sandbox method known as Software-based Fault Isolation [20, 17] restricts an object code from writing or jumping to a memory address outside of a separate portion of application’s address space called the fault domain. This is done by inserting into the binary code some runtime check before the store or jump instructions. Extension of the compiler [19, 6, 4] can analyze the code to enforce a safe use of the commonly exploited function of the C library \texttt{scanf()}, \texttt{strcpy}(...) . This prevents the attack like buffer overrun techniques by detecting vulnerabilities in the code. The major drawback of this approach is that this detection might be circumvented by self modifying code.

Interposition mechanism can be implemented by trapping the system calls level (Janus [12], Consh [2], MAPbox [1]) using the Ptrace facility. System calls are an abstraction by which a process access to the resources of a machine.

Ptrace is a mechanism provided by the linux kernel to let process take over control of its child whenever they perform specific actions in particular system calls. Any time a ptraced process do a syscall, it is frozen and its parent is signaled with a SIGCHLD signal. The parent can then read and write its child memory to determine what actions is requested by the frozen child. In a sandboxing context, this is used in the following manner: the parent process checks the parameters of the call before the call is performed. If they are compliant with the security policy, the system call is executed, otherwise the child can be killed, the parameters can be modified or any other corrective action can be carried out.

Another approach of kernel security presented in [7, 18] provides an enhanced interposition mechanism inside the sensitive functions of the kernel.

For this study we have chosen to experiment two Ptrace based sandboxes and one based on a kernel extension.

2.2.1 Subterfugue

Subterfugue [15] is a user space sandbox based on Ptrace and written in the python language. Security policies, called tricks, are written as Python classes for each one of the system calls that have to be controlled. Subterfugue launches the user program and catches every system calls of the sandboxed process. Thus for every system call, Subterfugue freezes the process and execute the corresponding python code, which can lead, in case of a security check failure, to interrupt its execution. If the security check is valid, the Subterfugue lets the process perform the system call.

2.3 User Mode Linux

User Mode Linux [9] is a patch for the linux kernel which allows it to run in user space on a Linux station. More precisely, the executable of UML is a user space program which simulates a complete Linux kernel, where one can execute programs.

Every task running inside the UML system is a thread of the UML application. It is Ptraced by the main UML thread i.e. the kernel. Every time the task does a syscall, UML catch it by the Ptrace mechanism and then, at the contrary of Subterfugue, simulates the syscall in user space. UML seems like a virtual machine, an autonomous Linux system, totally isolated from the hosting site. Used as a sandbox, the safety policy is provided by the configuration of the virtual machine that is for instance mapping of the virtual file system to a specific file on the execution site.

2.4 Linux Security Module

Linux Security Module [7] is an interposition mechanism at the kernel level. It is a patch for the linux kernel which add control code at the end or the beginning of the sensible functions of the kernel : the hooking code. The idea is that the system administrator can enhance these functions by the way of registering a module. This module contains functions that are called by the hooking code. They contains specific tests which provide an extended rights mechanism. As it is an open format, everybody can use his own security policy by coding the interposition functions.

The functions of the kernel which are concerned by the hooks are syscalls and some internal functions which are more generic than the syscalls. In fact, the interposition is made in a semantic way. The hooked functions are called when the system effectively performs a sensitive action.
This means that the interposition is driven by internal actions rather than user calls.

For example, the permission hook of the file system is called in eight different functions of the kernel: read, write, readv, writev, pread, pwrite, vfs_readdir and sendfile. From this only one check function is necessary for all these syscalls and LSM ensures that every time a file is manipulated, this hook will be called.

3 Performance Evaluation

In this section, we present an evaluation of different benchmarks under four security systems: Java virtual machine, User-mode-linux, Subterfugue and Linux security module.

3.1 Test Platform

The test platform is a dual Pentium 4 1700Mhz with 256KB of L2 Cache and 256Mo of RDRAM running Linux 2.4.9. We tested the different benchmarks on a set of different architectures and we found similar results, thus we only present results for this machine. The overhead due to the sandbox depends on the implementation of the security policy, in this study we have used the simplest security policy possible, in order to focus on evaluating the interposition cost. For Subterfugue and LSM the security policy allows all system calls and simply count them.

3.2 Benchmarking the sandboxed native code

3.2.1 I/O Micro Benchmarks (lmbench)

The lmbench suite is a commonly used toolbox of micro benchmarks to test and compare different UNIX systems. We used a subset of this tools to measure the overhead on commonly used syscalls which present an interest on the security point of view. getpid is the measure of the fastest syscall which return the pid of the calling process. read, write, stat, open and close are the standard file operations. fork+ exec /bin/sh is a composite call which executes a fork and execute the standard shell.

Table 1 presents for every sandbox technique, its execution time in $\mu$s and the slowdown (in percentage) produced by the use of the sandbox compared with a non-sandboxed execution of the same syscall.

Globally, Subterfugue is two hundred time slower on short syscalls, and up to twenty times for longer syscalls. This is mainly due to the ptrace mechanism which implies context switches between the parent and the child process. And other reason is the use of the python language (an interpreted language) to implement the security policy. UML suffers from the same Ptrace slowness even if it performs four time better.

The difference between LSM and original Linux is very tight because of the efficiency of the kernel hooks: no switch of context, just a kernel space function call by hook.

3.2.2 Performance evaluation on applications : gzip, gunzip and gcc

We present the execution time of three real applications which mix a high number of I/O operations on files and calculus. gzip and gunzip are two standard gnu applications. They performs respectively inflating and deflating of files. For the test, we measured compression and decompression time of a 542Mo file. gcc is the GNU C compiler. The test consists in the compilation of 70000 lines of C code extracted from the xfig source.

The measured time was computed by substracting the date of begining of each process from their completion time.

On the table 2 we present the execution time measured on our benchmarks depending on the sandbox system used to confine their execution, first in $\mu$s and after in percentage based on non-secured execution.

Subterfugue has here better performance than UML. In fact, even if the hooking treatment of Subterfugue is slower than UML one, it costs a lesser part of time because there are less context switches. Moreover, as UML is a complete system, more threads are running and then the benchmark process have lower chance to be scheduled.

Another non intuitive result is that LSM is a little faster than native linux. This come from a particularity of the LSM patched kernel. The part concerning the capabilities become a module inserted in LSM as a particular security policy. This security policy can not coexist with another policy. Thus, the capability related tests are not executed in LSM which gives a small acceleration.

The Table 3 presents the distribution of the system calls for the three benchmarks. This shows that the Ptrace based solutions are more penalized a high frequency of syscalls done by the application. Moreover for gcc, a large number of syscalls are performed which are not sensitive from a security point of view (brk, fstat64, mmap).

3.3 Benchmarking the Java Virtual Machine

3.3.1 Performance evaluation on numerical benchmarks

We compare the performance of a language based security system (Java) versus a native code execution using four Java Virtual Machines. In fact for pure numerical benchmarks we have not found significant difference between a native execution without security or inside a sandbox.
Table 1. Measure of system calls execution time in $\mu$s inside three native code sandboxing environments and comparison expressed in percentage with their execution time without security.

<table>
<thead>
<tr>
<th></th>
<th>getpid</th>
<th>read</th>
<th>write</th>
<th>stat</th>
<th>open/close</th>
<th>fork+bin/sh</th>
</tr>
</thead>
<tbody>
<tr>
<td>without sandbox $\mu$s</td>
<td>1.03</td>
<td>1.36</td>
<td>1.27</td>
<td>4.96</td>
<td>7.05</td>
<td>4644</td>
</tr>
<tr>
<td>LSM in $\mu$s</td>
<td>1.03</td>
<td>1.37</td>
<td>1.29</td>
<td>5.17</td>
<td>7.29</td>
<td>4474</td>
</tr>
<tr>
<td>LSM in %</td>
<td>100</td>
<td>99.3</td>
<td>98.4</td>
<td>95.9</td>
<td>96.7</td>
<td>96.3</td>
</tr>
<tr>
<td>Subterfugue in $\mu$s</td>
<td>214.04</td>
<td>249.70</td>
<td>251.46</td>
<td>251.42</td>
<td>489.35</td>
<td>73759</td>
</tr>
<tr>
<td>Subterfugue in %</td>
<td>0.48</td>
<td>0.54</td>
<td>0.51</td>
<td>1.97</td>
<td>1.44</td>
<td>6.30</td>
</tr>
<tr>
<td>UML in $\mu$s</td>
<td>51.64</td>
<td>52.45</td>
<td>51.88</td>
<td>73.01</td>
<td>126.56</td>
<td>95596</td>
</tr>
<tr>
<td>UML in %</td>
<td>1.99</td>
<td>2.59</td>
<td>2.45</td>
<td>6.79</td>
<td>5.57</td>
<td>4.86</td>
</tr>
</tbody>
</table>

Table 2. Execution time of three benchmarks in $\mu$s confined in sandboxes and comparison in percentage with the same execution without security.

<table>
<thead>
<tr>
<th></th>
<th>gzip</th>
<th>gzip</th>
<th>gunzip</th>
<th>gunzip</th>
<th>gcc</th>
<th>gcc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(s)</td>
<td>(%)</td>
<td>(s)</td>
<td>(%)</td>
<td>(s)</td>
<td>(%)</td>
</tr>
<tr>
<td>Without security</td>
<td>386</td>
<td>100</td>
<td>183</td>
<td>100</td>
<td>74</td>
<td>100</td>
</tr>
<tr>
<td>LSM</td>
<td>383</td>
<td>100.7</td>
<td>179</td>
<td>102.3</td>
<td>70</td>
<td>105</td>
</tr>
<tr>
<td>Subterfugue</td>
<td>408</td>
<td>94.5</td>
<td>185</td>
<td>99.4</td>
<td>165</td>
<td>45</td>
</tr>
<tr>
<td>UML</td>
<td>462</td>
<td>83.6</td>
<td>214</td>
<td>85.7</td>
<td>172</td>
<td>43</td>
</tr>
</tbody>
</table>

Table 3. Characteristics of the benchmarks in term of distribution of the system calls.

<table>
<thead>
<tr>
<th></th>
<th>Number of system calls</th>
<th>Syscalls per seconds</th>
<th>fork (%)</th>
<th>open/close (%)</th>
<th>read/write (%)</th>
<th>other (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>26671</td>
<td>70.1</td>
<td>0</td>
<td>0.03</td>
<td>99.86</td>
<td>0.11</td>
</tr>
<tr>
<td>gunzip</td>
<td>22032</td>
<td>120.4</td>
<td>0</td>
<td>0.04</td>
<td>99.81</td>
<td>0.15</td>
</tr>
<tr>
<td>gcc</td>
<td>136078</td>
<td>1943</td>
<td>1.32</td>
<td>44.7</td>
<td>9.25</td>
<td>44.73</td>
</tr>
</tbody>
</table>

The benchmarks used are FFT (One-dimensional Fast Fourier Transform), SOR (Jacobi Successive Over-relaxation), MatMul (Sparse matrix vector multiplication), MC (Pi calculation with a Monte Carlo integration) from the Scimarks2.0 benchmarks suite written in C and Java language, MolDyn (molecular dynamic N-body code) from the Java Grande Forum Benchmarks suite provided as Java code, we had translated in C, LU (LU Matrix factorization) from the Linpack benchmark available in Java, C and Fortran and finally EP and CG from the NAS serial benchmark originally written in Fortran and that we have translated in Java. While EP and CG are kernels in the NPB benchmarks suite, there are more complex than the previous benchmarks. For the Scimarks benchmarks the large size of the problems has been used, and the NAS kernels (EP, CG) are class A sized.

For the following tests we have also used a familiar computer representative of the mean performance expected in a Peer-to-Peer system. This computer is powered by an Intel Pentium III 733Mhz, 256KB L2 Cache and 256 MB of DDR-SDRAM.

We have tested four Java environments representative of two virtual machine technologies: SUN JDK 1.2.2 is a just-in-time compiler, HotSpot SunJdk 1.3.1, SunJdk 1.4.0 beta and IBM JDK 1.3.0 are adaptive virtual machines. Native compiler used were GNU gcc 3.0.1 and gcc 3.1 (17-09-2001 CVS Snapshot) with the strong and processor dependent optimizations for Intel Pentium 4 with extended instruction set (SSE2), Intel C/C++ 5.0 icc and Portland Group Compiler Fortran compiler pgf77. For the Java benchmark, the compilation has been done using -O flag.

We have done 10 runs for every benchmarks, and present in the figure1 and 2 the best execution time. For the native code, used as base unit, we have selected the fastest execution among the different compilers tried with various optimizing directives. Compiler and flags used for the code generation are presented in the table 4.

The fastest JVM is the IBM Jdk 1.3 with an average execution time close to the native code (ranging from 98% to 66%) except for the MC benchmarks on the both machines. The slowest JVM is the Sun jdk1.2 with a worse case execution time 10 times slower than the native code. It is interesting to note that the underlying compilation mechanism has been drastically improved between jdk 1.2 and the next
generation jdk1.3. Native code is always faster than bytecode execution even if the gap is reduced with new JVM generations.

For the two figures the base is the execution time of the native code. From a global point of view, we notice that all the java runtimes provides worst performances compared to the native execution, but the ratio is dependent on the java implementation. As for the IBM runtime, the perfor-
mances are nearly the same as the native execution on some benchmarks. The same tendance can be viewed on the both figure 1. Of course, some runtimes performs better depending on the architecture (for nearly all the benchmarks), but with few exception we keep the same tendance. We have to notice the poor results on the MonteCarlo benchmark. A possible explanation is that there are a big number of embedded calls that seems to be executed one by one by the java runtime but are inline by the C compiler. On the figure 2 we present the comparaison between the execution time on both processor architectures. We can discern that the java runtimes profits better on the P4 architecture than the native execution, although all the executions scale with the processor performances. Anyway we are still far from similar execution times between java and native.

3.3.2 Performance evaluation on I/O Micro Benchmarks

We have adapted the lmbench MicroBenchmark to the Java environment in order to evaluate the performance of several JVM on I/O Benchmarks. The experience considers the case of a JVM running with and without security manager. The security manager uses a straight security policy where the file access in open, read and write mode recursively over the /dev directory is granted to the benchmark code.

Figure 3 presents the mean execution time of basics file I/O operations for four JVM. The performance of the JVM without a security manager is to be compared with raw performance of the native code. In this case we observe that java suffers of an important slowdown mainly due to open/close operations (38 μsec versus 7 μsec) while the read and write operations have closer performance to the native code (1.72 μsec versus 1.25 μsec for read operations). As the java security manager checks the permissions only on open and close operations, they performs slower (the open/close operation take 138 μsec), similar to UML but 5 time faster than subterfugue. Note that for Java, Read and Write operations are not affected by the security Manager.

4 Qualitative evaluation of the sandbox security

In this article, we do not evaluate the quality or the cost of a real security policy. We assume that implementing such a policy has the same cost for every kind of system. Thus we focused our study on the cost of the interposition mechanism. Beside this cost, there are qualitative arguments for every evaluated systems. In this discussion, we try to pinpoint the drawbacks and advantages of every methods.

Supporting native code presents a lot of advantages. First, there are lots of legacy codes written in many different languages: running native code permits massive reuse of them. Moreover, it permits to choose the compiler and compilation options.

Sandbox based on Ptrace suffers from three major drawbacks.

- The first one is a race condition. The problem comes from the context switching inherent to user process nature of the monitor (parent). Using the parallelism, the child can change the arguments of the call just between the monitor validation and the real call performed by the kernel.

- The second drawback is that a user process can perform a syscall without being ptraced. Linux provide a functionality called personality which helps in running code designed for other UNIX architectures. This is implemented by a syscall named setpersonality which allow a process to use another set of syscalls which are not ptraced (the so called lcall7 syscalls).

- The last drawback is the poor performance of that technique: every time the sandboxed program performs a syscall, we pay the cost of at least two contexts switching.

Subterfugue is a light tool designed for trivial security goals. It’s strength comes from the fact that it is a completely user space solution. On the other hand, it suffer from all ptrace drawbacks, including that it is not completely secure.

UML is a good solution but not directly dedicated to sandboxing. For now, UML is not completely secure: there are some pages of memory, dedicated to the UML kernel which are not protected, and accessible by the others UML threads. Unfortunately, by this way it will be possible for a sandboxed application to migrate out of the sandbox. According to the authors, this will be fixed before first stable release. The major profit of UML is that it is a complete system, which permits to control every system component including: memory allocated, disk space (without quotas), kernel and libraries versions. It is also a solution which does not require super user privileges. These two points are determinant in a peer to peer global computing environment.

LSM is the best solution for execution of native code in terms of performance. It is also the solution that addresses the problem at the lower level which guarantes the best security level. LSM is born from the necessity to have a modular and unified security mechanism in Linux. Thus, this project will eventually merge with the official linux tree.

The major drawback of LSM is its very low level of interposition: this means that it must be installed as a new kernel/module and this operation cannot be achieved by a normal user. More than that, it is very risky to give such a control to a security policy that is not easily checkable.
Another interesting feature of LSM is that it allocates kernel memory in the objects structures. Then, the policy fills this memory with specific security informations which is used later by the other hooks.

For example, every time an inode is looked up (creating an inode structure which corresponds to a file), the function post_lookup is called to let the policy allocate and initialize some memory in the inode structure. This information will be used later by the file related hooks without any new file operation. This memory is in kernel space, thus it is unaccessible to any user space program.

According to our results, the IBM Jdk is able to approach native code performance. Combining with the code portability and the built-in security mechanisms, Java might appear as a plateform of choice for Peer-to-Peer Computing. It allows to implement fine grain security policy, eventually different according the provenance of the mobile code. But the ignorance, for the provider of a code, of the JVM installed on the client side may lead to unexpected performance. Another issue is that the use of Java byte-code is exclusive with the use of native code. In fact the call of native code throught shared libraries completely breaks the security model of the JVM.

5 Secured execution environment in a Peer-to-Peer system

This section describes the integration of a secured execution environment in a Peer-to-Peer system: XtremWeb. We evaluate the overhead issued by the use of a sandbox by running a parallel benchmark EP ported from the NAS NPB-2.3 on the Peer-to-Peer system.

XtremWeb [10] is a framework for experimenting with Global and Peer-to-Peer Computing. XtremWeb follows a push/pull paradigm, entities submit jobs to the system, while others entities, on their idle time request jobs, execute them and send back their results. More precisely, a job consists of a mobile code, which is provided either as native code or as Java byte-code and a set of parameters. Usually applications are sequential, as XtremWeb in its current version targets multi-parameters studies. Their parameters are provided as a set of compressed files and as a command line arguments. When a job is retrieved by a remote resource, first the parameters are deflated in a temporary directory. If the mobile code is native, a process is launched and the temporary directory is set as its current working directory. XtremWeb itself is written in Java therefore if the mobile code is a Java byte code, the byte code is loaded in the same instance of the virtual machine than XtremWeb itself, and the method corresponding to main entry (described in the manifest of the byte code archive) of the application is invoked.

For native code we have integrated the Subterfugue sandbox. To execute Java byte code, a custom class loader creates a protection domain for the mobile code and assigns it the permissions specified by the security policy. In both cases, the security policy implemented is very restrictive. The mobile code does not have any access to the network and to the file system. The only right granted is the possibility to access in read/write/delete modes the files in the temporary directory specially created for the duration of the job.

The experiment seeks to evaluate the relative performance limits of sandboxed and non sandboxed executions on a real peer-to-peer infrastructure. To evaluate the performance of the different approaches we have adapted the EP kernel from the NAS NPB-2.3 parallel benchmark suite to the XtremWeb peer-to-peer system. Such kind of benchmark is not suited for large peer-to-peer computing system especially because its ratio memory occupation/computing time is very high. For example, only the Class A benchmark can be executed on usual off-the-shelf PC. The execution time of EP for this class rapidly becomes very low when the number of processors increases. The typical execution time of EP Class A on 10 Pentium III PC running at 500 Mhz is about 25 seconds. This time is approximately the overhead of the XtremWeb system to handle 10 tasks (launching tasks, packing, encrypting and sending the
parameters, retrieving, unpacking the results). So this experiment does not provide any results about the merit of the peer-to-peer infrastructure itself. The only purpose of using EP is to illustrate the impact of the system overhead on the relative performance of the various secured and non-secured executions.

We have started from the MPI implementation of EP, and keep the parallelization in a master/slave style. The master initiates the EP computation, spawns several jobs and submit them to the XtremWeb system. As there are no means to directly communicate between the nodes, the parameter of the jobs are passed through the command line. Those parameters are the total number of spawned jobs and the job number, which respectively corresponds to the MPI_WorldSize, and MPI_MyRank of the MPI implementation. For each created job, XtremWeb returns a handler used to retrieve the results of the job when it will be completed.

For this experiment we have used a processor farm with 16 Pentium III interconnected with ethernet 100Mbits running the slave part and one node running the master part. The execution time measured starts before the submission of the first job and ends after the reception of the last result.

Figure 4 present the execution time of EP benchmark, on 16 nodes, with a varying number of jobs. The leftmost part of graph shows a better performance for an execution of EP native code than an execution in byte code. At the contrary, there is no penalty for using a sandboxed environment. This can be easily understood as EP does not perform any file operations and thus does not trigger the sandbox security checks. After 16 tasks, the curve reaches a minimal point. This means that the general overhead of the peer-to-peer system prevents the benchmark to achieve a better scalability. This overhead is mainly due to the high communication cost as secured and encrypted sockets are used, and intermediate copies of the results. This result should be considered as a general trend for Peer-to-Peer computing systems: difference between native and Java performance, sandboxed and non sandboxed execution performance are lighten by overhead of the distributed system management.

Note for the reviewers: we will present results with an out-of-core benchmark, the polyphase merge sort, for the final version of the paper.

6 Conclusion

Global Computing and Peer-to-Peer computing harvest the idle time of computers provided by volunteers connected to the Internet. Peer-to-Peer computing has two fundamental characteristics. One is that every node of the network can act both as a computing resource and as a provider of code. The other is that due to the very large scale of resources no authentication mechanism can avoid users to spread hostile code. A solution to this problem is the use of sandboxing: a system that confines execution of a code in a safe environment.

In this paper we have presented an evaluation of performance of three sandboxing mechanisms that allow native code execution and four Java Virtual Machine which execute byte-code.

The native sandbox we have evaluated follows two approaches for interposition: Subterfugue and User Mode Linux use the Ptrace mechanism provided by the Linux kernel in user space and Linux Security Module uses an interception mechanism at the kernel level.

Performance evaluation on micro-benchmarks shows that the overhead for system calls is high for the two Ptrace based systems and insignificant for the kernel based system. Confronted with three applications that mixes calculus and numerous input/output operations, Ptrace based systems can reach a 43% slowdown while the kernel based system doesn’t induce any overhead. Performance evaluation of language based sandbox were conducted using four different Java Virtual Machines. On numerical benchmarks, we confirm that native code execution outperforms Java byte code while the performance of the new generations of JVM are getting closer to the native code execution.

We have shown that considering a real Peer-to-Peer environment and limit conditions, the performance differences between native/Java executions and sandboxed/non sandboxed executions are reduced due to the overheads related to the distributed system operations.

Beside this performance issue, we think that the trade-off for choosing a sandboxing environment will be on flexibility/performance. User-Mode-Linux offers a complete virtual system detached from the hosting system while Linux Security Module offers the best performance and likely the safer protection. Choosing Java as a sandbox solution might strongly depends on the amount of legacy code to rewrite.

References


Figure 4. Performance evaluation of the execution of EP on a Peer-to-Peer system with and without a sandbox in native ix86 code and in Java byte code.


[13] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going beyond the sandbox: An overview of the new security architecture in the java development kit


