Olympiads in Informatics, 2012, Vol. 6, 100–109

A New Contest Sandbox

Martin MAREŠ[1], Bernard BLACKHAM[2]

[1]*Department of Applied Mathematics, Faculty of Mathematics and Physics*
Charles University in Prague
Malostranské nám. 25, 118 00 Praha 1, Czech Republic
[2]*School of Computer Science and Engineering, University of New South Wales*
Sydney NSW 2052, Australia
e-mail: mares@kam.mff.cuni.cz, bernardb@cse.unsw.edu.au

Abstract. Programming contests with automatic evaluation of submitted solutions usually employ a sandbox. Its job is to run the solution in a controlled environment, while enforcing security and resource limits. We present a new construction of a sandbox, based on recently added container features of Linux kernel. Unlike previous sandboxes, it has no measurable overhead and is able to handle multi-threaded programs.

Key words: automatic grading, sandbox, containers, threads, computer security.

1. Introduction

Many programming contests in the world employ automatic grading of programs submitted by contestants. This is usually accomplished by running the submissions on batches of input data and testing correctness of the output. The program must also finish each test run within given time and memory limits, so that it is possible to distinguish between correct solutions of different efficiency.

Additionally, proper security measures must be taken to avoid cheating – e.g., the program must not be allowed to access files to steal the correct answer, kill other processes, nor communicate over the network.

To achieve both security and limits on resources, programs are usually run within a controlled environment called a *sandbox*. In recent years, most programming contests seem to be run on Linux systems, so we will study Linux sandboxes only.

By far, the most common kind is a tracing sandbox; see, e.g., Mareš (2007) or Kolstad (2009). It uses the `ptrace` system call to ask the kernel to stop the sandboxed program whenever it is about to execute a system call. A monitoring process then examines the arguments of the system call and either lets the call proceed, or kills the program for committing a security violation. Time and memory limits are usually enforced by a combination of system call monitoring and the standard resource limit infrastructure inside the kernel (the `setrlimit` system call).

Recently, the overhead of system call tracing has received lots of attention. Merry (2010) has shown that especially in the case of interactive tasks, the number of system

calls can become very large and so can the time spent monitoring them. Mareš (2011) has published a more careful analysis, which shows that while the overhead is significant, it does not affect contest fairness much. He also evaluates several techniques for decreasing variance of time measurements, like pinning of tasks to specific processor cores, or real-time scheduling.

Another common problem with ptrace-based sandboxes is that they are unable to enforce security of programs running in multiple processes of threads. At the first sight, it seems not to matter, since programming contests rarely involve writing parallel programs. However, the run-time environments of several programming languages automatically create several service threads. This happens for example in the Java Virtual Machine, and in Mono, which is an implementation of the Microsoft Common Language Infrastructure.

Ptrace-based sandboxes are also highly architecture-specific. Supporting both 32-bit and 64-bit execution environments on x86 requires different implementations of several aspects of the sandbox, including the list of allowable system calls. This is complicated even further on 64-bit x86 machines, as the evaluated program may be either a 32-bit or 64-bit binary, or even a 64-bit binary using 32-bit system calls.

In this paper, we present a new design of a contest sandbox. It is based on namespaces and control groups, which are new features of the Linux kernel. These are primarily intended for partitioning a large machine with many processors into multiple nodes, but have turned out to be useful for our purposes, too. The advantages of this approach include a much smaller overhead and the ability to reliably sandbox multi-threaded programs.

In Section 2 we give a brief overview of some related state-of-the-art sandboxing techniques. In Section 3 we outline our requirements of a sandbox for programming contest environments. In Section 4 we describe the kernel features which we use for our sandbox, followed by a description of our implementation in Section 5. Finally, we measure the performance overhead of our sandbox in Section 6.

2. Related Work

Research on code isolation did not stop with simple tracing of system calls. Let us review currently known sandboxing techniques first.

Merry (2009) has suggested a contest sandbox based on the Linux Security Module infrastructure. Instead of monitoring the system calls by a user-space program, it inserts a kernel module, which uses kernel security hooks to check parameters of system calls. The time and memory overhead is almost zero, but Merry's implementation does not support multiple threads and it would be complicated to do that. The main drawback of this approach is the instability of the security module interfaces between kernel versions – thus the module has to be updated for every new kernel release.

It is also possible to use hardware virtualization or para-virtualization, so that the contestant's program can run on its own virtual machine with its own instance of the operating system (OS). All interaction between the program and the rest of the world can be easily limited by the configuration of the virtual machine. Unfortunately, the overhead

of contemporary virtual machines is very high and suffers from a large variance, so it does not seem possible to use them in a fair contest.

Software fault isolation (SFI) is a method of enforcing the secure execution of arbitrary code, as described in Wahbe *et al.* (1993). SFI is typically used to provide isolation between modules running in the same address space (e.g., dynamically loaded plug-ins). This approach is overkill for batch-processing type tasks used by many contests, as a contestant's program executes in a separate address space, enforced by the processor. It could be applicable to interactive tasks where the context switching overheads are high. However, many of these techniques have high overheads, and require recompilation of code using special compilers to intercept all potentially harmful operations.

Native Client (NaCl) is another approach to sandboxing native executables proposed by Yee *et al.* (2009). It implements software fault isolation, performing static analysis to ensure that all user-supplied code can be safely executed, but also makes use of hardware support to enforce memory protection. Once checked and loaded, user code runs at native speed. NaCl enforces specific requirements on the executable's structure to ensure the static analysis is sound. Like other SFI-based approaches, binaries must be compiled with specific compilers and libraries.

Linux also offers a security module known as *seccomp*, which can limit the system calls accessible to a program. This is similar in spirit to ptrace-based approaches, but in its original version, it was limited to allowing a very restrictive, fixed set of system calls. Programs had to be significantly modified to execute under seccomp. More flexible forms of seccomp have been proposed in the past, but as of the time of writing, none of these have made it into the upstream Linux kernel.

TxBox, developed by Jana *et al.* (2011), supports sandboxing of arbitrary processes through the use of operating system-level transaction support. Using transactions can limit the impact of untrusted insecure code, as system state can be "rolled back" after execution. This provides strong isolation properties and can work with arbitrary executables, but requires significant modifications to the OS kernel, and is not (yet) supported in popular Linux distributions.

3. Requirements for a Contest Sandbox

Most programming contests require students to submit the source code for their solution, which is automatically compiled and graded on a server. We wish to minimize the work required for a contest organiser in setting up such a server, and as such we aim to support a modern unmodified Linux distribution. We also wish to support as many languages as is practically feasible. This precludes the use of special customized OS kernels, distributions or custom language-specific toolchains. We aim to sandbox binaries compiled using standard compilers, and still ensure the integrity of the system.

In contest environments, the only untrusted part of the system is the program submitted by the contestant. All security risks therefore involve interaction between this program and the rest of the system. Any such interaction requires the use of system calls. Let us

review the list of all Linux system calls and look for their potential security issues. (We refer the reader to Kerrisk *et al.* (2012) for a concise description of Linux system calls and to Kerrisk (2010) for a more comprehensive treatise on Linux system interfaces.)

We note that our list includes only the security risks known to us (including those mentioned by Forišek (2006)) and we do not have a formal proof that it is complete. (In fact, such a proof would necessarily include a proof of correctness of the whole Linux kernel.)

The problematic system call groups are:

- *Access to files* (`open`, `read`, `write`, `stat`, ...). The set of files available inside the sandbox can be easily restricted by using traditional filesystem permissions. The sandboxed program runs under its own user and group ID and all directories outside a designated area are made inaccessible or read-only for that user ID. Additionally, we can change the root directory of the process to some sub-directory. We must not forget that the amount of data written to disk by the program has to be limited, which can be achieved using disk quotas.
- *Allocation of memory* (`brk`, `mmap`, `mlock`, ...). We must restrict the amount of memory available to the program. Even if we do not want to grade memory complexity of solutions, we must avoid exhaustion of system memory, which could cause the other parts of the contest system to fail. In case of a single process, `setrlimit` can be used to limit the total amount of address space available. There is no traditional UNIX mechanism for limiting total memory consumption of a group of processes.
- *Creation of processes and threads* (`fork`, `clone`, `vfork`). In a traditional Pascal/C environment, only a single process or thread (Linux does not distinguish between them internally) should be allowed. If we want to support multi-threaded runtimes, we must place a limit on the total number of processes to avoid overloading the task scheduler. Fortunately, there is a system resource limit on the total number of processes run by a given user. Also, a program may attempt to evade time limits by splitting the calculation into several processes, each running on different physical processors. To prohibit this, we have to measure the sum of execution times of all processes within the sandbox.
- *Sending of signals* (`kill`, `killpg`, `tkill`, ...). Signals are asynchronous events delivered to processes or threads. Sending of signals to our own process is harmless, but we must forbid sending signals to other processes. Otherwise, a part of the grader or even an unrelated process can be killed or confused. As expected, UNIX already disallows signalling processes run by other users, so running the process under a unique user ID suffices.
- *Inter-process communication* (`shmget`, `msgget`, `mq_open`, ...). There are several APIs for sending messages and sharing parts of memory. They are controlled by filesystem-like permissions, but a process can create a message queue or shared memory object accessible to everybody. This could be used for cheating, so we want to forbid such operations, or even better put a barrier between communication objects of our process and the rest of the system. Alas, there is no traditional mechanism for that.

- *Networking* (`socket`, `bind`, ...). All attempts to communicate over the network must be stopped. This includes not only TCP/IP networking, but also other address families, including local sockets addressed as a part of the filesystem. The traditional UNIX socket API does not offer any configurable limits.
- *Executing other programs* (`execve`). The program can run other applications and let them handle a part of the competition task. For example, it could replace calculations with big integers by calls to `bc` or `python`. While we do not necessarily consider such tricks unfair, we would still like to support contest organizers who do so. We do not see a way how to disallow `execve`, but it is easy to keep the directory tree available inside the sandbox free of all foreign programs. Optionally, some parts of the directory tree can be mounted with the `noexec` option, so that binaries stored in them cannot be executed at all.
- *Sleeping* (`pause`, `sigsuspend`, `wait`, `nanosleep`, `poll`, ...). There are multiple system calls which suspend the program until some event occurs. This may be for example an arrival of a signal, exit of a child process, readiness of data on a file descriptor, or simply the expiration of a timer. None of these operations compromise security, but they can lock up the grading system for an indefinite amount of time. To avoid that, we limit not only the execution time, but also the time elapsed on a "wall clock" (i.e., an independent clock measuring real time). The wall-clock time limit is usually set somewhat higher than the run-time limit, so the program does not time out if it shares the processor with other programs.
- *Accessing system time* (`alarm`, `gettimeofday`, ...). The rules of some contests explicitly forbid reading of system time. This limitation is hard to support without explicit system call tracing and we do not consider it important as there are many side-channels which can be used to estimate elapsed time. One example is the time-stamp counter (TSC) inside the processor, which is essentially a register containing the number of CPU cycles since system boot. We have therefore decided not to limit time-related system calls.
- *Flushing buffers to disk* (`sync`, `fsync`, `sync_file_range`, ...). These operations do not compromise any secrets, but can result in added disk I/O activity. This extra disk activity can slow down the execution of both the contestant's program and other processes on the system. In our opinion, imposing wall-clock time limits is sufficient to limit the effects. For those contest administrators who are still concerned, one possible work-around is to put the writeable part of the directory tree on a RAM-disk.

4. Kernel Compartments

Traditional virtualization focuses on running multiple instances of Linux using a hypervisor, whereas OS-level virtualization (or compartments) offers lower overheads and greater timing predictability. There exist several projects to add OS-level virtualiza-

tion into the Linux kernel, including OpenVZ[1], LXC[2], and the Linux-VServer project[3]. Thanks to these projects, there is now sufficient infrastructure in the official Linux kernel to support compartments for running arbitrary untrusted code.

4.1. *Namespaces*

The Linux kernel provides the ability to isolate groups of processes through the use of *namespaces.* A namespace is a context used for specific kernel operations, such as networking, filesystem access or process control. Specifically, each Linux process belongs to:

- *a process namespace:* this is the set of all processes which are visible to a compartment. An isolated process should live in its own empty process namespace, preventing it from signalling or otherwise interfering with any other process on the system. Process namespaces form a hierarchy when created, such that a process is visible in its own namespace and all parent namespaces. Furthermore, when the top-level process of a namespace exits, all other processes spawned by it are automatically terminated. (Without this feature, reliably terminating a hierarchy of possibly malicious processes is almost impossible.)
- *a networking namespace:* this determines the set of networking devices available to the compartment, which for isolated processes should be none. Although isolated processes can still create IP sockets, they cannot use them as there are no network devices (not even the loopback interface).
- *a filesystem namespace:* this defines the set of mounted filesystems that can be accessed by processes. An isolated process may have its accessible filesystems stripped down to the bare minimum – this may simply be a single RAM-disk. Unlike traditional chroot jails, filesystem namespaces cannot be easily circumvented.
- *an IPC namespace:* this namespace protects the inter-process communication system calls which provide shared memory and message-passing (`shmget`, `msgget`, `mq_open`, ...). In a typical UNIX system, all processes on a machine may potentially communicate. By placing an isolated process in an empty IPC namespace, it is unable to interact with any other processes on the system using IPC.

When a new process is created using the `clone` system call, it can be optionally placed into new empty namespaces in each of the above categories (or otherwise inherit its parent's). These can be used to effectively limit what kernel resources are accessible to a process.

4.2. *Control Groups*

In the previous section, we showed that namespaces can tightly control access to kernel-provided resources and limit interaction between processes. However, they do not enforce any control over CPU time or memory resources. Traditional UNIX mechanisms

[1] `http://www.openvz.org/.`
[2] `http://lxc.sourceforge.net/.`
[3] `http://www.linux-vserver.org/.`

only allow CPU and memory to be limited on a per-process basis. They do not scale for enforcing these limits for groups of processes.

Linux has recently introduced the concept of control groups to achieve this. The administrator can define a hierarchy of control groups and place processes at arbitrary points within it. New processes automatically inherit their parent's control group.

A hierarchy can have several *controllers* connected to it. A typical controller manages some resource. For each group, it tracks the usage of the resource by all processes inside the group and its subgroups. Each group can also have its own limits on the maximum allowed usage.

The following controllers are interesting for our purposes:

- *CPU set controller:* processes inside a group can be tied to a subset of available processors, processor cores or memory nodes. We want to reserve such a subset for exclusive use by the sandbox. This helps us minimize timing noise caused by context switches and related caching effects. This timing noise can add significant variance to the measured execution time, as shown by Merry (2010).
- *memory controller:* total memory used by the group can be tracked and limited. This includes not only explicit memory allocations by processes, but also everything implicitly allocated by the kernel on behalf of these processes, for example cached parts of files.

  The accounting of memory pages is complicated by the presence of pages shared between processes belonging to different control groups. The memory controller accounts for them on one more-or-less randomly chosen group. Fortunately, we have already limited inter-process communication, so the only shared pages which can really occur are parts of commonly used files (e.g., shared libraries). When the memory gets tight, such pages are reclaimed, so while they can affect reported memory use of the sandbox, they should not influence memory limits.
- *CPU accounting controller:* this controller does not impose any limits. It just tracks total CPU time used by processes inside the group. Unlike traditional task timers which are based on statistical sampling, this controller works with nanosecond-precision timestamps of context switches used internally by the process scheduler. We periodically monitor the CPU time used and when it exceeds the limit, we terminate all processes.

5. Implementation

We have implemented a new sandbox based on the ideas described above and checked that it withstands all attacks on security known to us.

The sandbox has been incorporated in the Moe modular contest system (see Mareš (2009) for an overview), but it does not depend on any other modules, so it can be easily used in other contest systems, too. Source code and a test suite are available at `http://www.ucw.cz/moe/`.

Please note that a recent Linux kernel is required (we have used version 3.2.2) and that it must be configured to support namespaces and control groups. This is not always the

case with default kernels supplied by Linux distributions. We refer to the documentation accompanying the source code for more details on system configuration issues.

5.1. *Features*

By default, our sandbox runs in a light-weight mode, which uses only namespaces and user identity separation to achieve security. In this mode, programs are limited to a single process or thread only. (In fact, more processes can be allowed, but time and memory limits then apply to each individual process, which is seldom useful.)

When use of control groups is switched on, the sandbox uses the CPU and memory controllers to limit overall consumption of resources. In this mode, an arbitrary number of processes and threads may be run.

Both modes support time and memory limits. There are separate time limits for real execution time and wall clock time. It is possible to keep the program running for a while after the time limit expires, so that the exact run time is known even for programs which have timed out. Memory limits in the control-group mode affect the total memory usable by both by the OS and processes, while in the light-weight mode they affect all virtual memory allocated by the process only.

All processes are started with a custom root directory stored in a RAM-disk. This directory contains only a set of mount points used for bind-mounting selected portions of the system directory tree. These usually include standard libraries mounted read-only and a read-write working directory, which holds input and output files of the program and it is limited by a disk quota. The contents of the custom root are configurable.

Unlike ptrace-based sandboxes, we do not require any knowledge of the CPU architecture. Our sandbox uses only isolation primitives provided by the Linux kernel, and is therefore portable to any host CPU architecture that is supported by Linux. This also avoids the complications associated with 32-bit and 64-bit executables, described earlier.

5.2. *Achieving Better Reproducibility*

As Linux is a very complex system, running on inherently complex hardware, there are an exponential number of states a system may be in when judging a solution. The state of the system when a program commences can directly affect not only the execution time of that program, but also its behaviour. There are a number of measures that must be taken in order to improve the consistency of the system across executions. They are essential to the fairness of any contest environment, not only of our sandbox. (See Mareš (2011) for further discussion.)

Address-space randomization is a feature used to protect system binaries from code-injection exploits. It does this by altering the address-space layout of a process each time it is started. Although this poses no concern for correct programs, buggy programs with memory management errors may behave inconsistently as the bugs might occur only for certain address-space layouts. This is clearly undesirable for a programming contest, and can be turned off by writing `0` to `/proc/sys/kernel/randomize_va_space`.

Table 1

The measured execution times of a process performing 1,000,000 system calls under different sandboxing techniques. The results are the average (and standard deviation) of 50 executions

Sandbox	Execution time [seconds]	Slowdown against native
None	3.56 (0.014)	–
Ptrace	9.26 (0.015)	160%
Namespaces	3.56 (0.010)	0%

Modern computer systems are designed to save energy when unutilized. One of the ways in which this is achieved is by scaling the CPU's frequency to best suit the current workload. This directly affects the run-time of processes and its behaviour depends heavily on what other activities are executing on the system. For a contest environment, frequency scaling must be disabled to ensure consistency across executions. This can be done on Linux by forcing the CPU to the highest available frequency, by writing `performance` to `/sys/devices/system/cpu/cpu*/cpufreq/scaling_governor` for each CPU on the system.

6. Evaluation

We compare the improvement in run-time of our sandbox over a traditional ptrace-based sandbox as used in Moe. Processes that perform very few system calls have negligible impact from either sandboxing approach. The overheads of ptrace-sandboxing are only seen when many system calls are performed. The results for a process performing 1,000,000 system calls are shown in Figure 1. These experiments were performed on an Intel Core-2 Duo T8300 CPU running at 2.4 GHz, using a 3.2.2 Linux kernel.

We measured the overhead introduced by a ptrace-sandbox to be 5.6 μs per system call. For many batch tasks, where the number of system calls performed is in the order of 10,000 at the most, this is negligible. However, interactive tasks which communicate with other processes could easily demand over 1,000,000 system calls. Here, the ptrace sandbox would add over five seconds to the evaluation time.

In contrast, we measured no overhead using our sandbox and all standard deviations were less than 0.5% of the mean. This is not surprising, as all isolation is performed inside the kernel using its standard mechanisms.

7. Conclusion

We have described a new type of sandbox, primarily intended for, but not limited to, use in programming contests. Unlike other sandboxes, it is able to isolate multiple processes and has very small overhead, while its implementation is very simple and architecture-independent.

Thanks to the small overhead, it gives significantly more precise execution timing for programs involving lots of system calls, especially for interactive contest tasks.

Support for multiple processes allows us to evaluate programs written in programming languages with multi-threaded runtimes, like Java, C#, or Erlang. Furthermore, it can be easily used to isolate execution of compilers and graders.

We are aware that the mechanism we use is not completely secure with respect to new kernel versions. A new kernel may include additional system calls to operate on new types of kernel objects, which do not belong to namespaces known to the sandbox. We however expect that such objects will be accompanied by new namespaces, so it will be possible to trivially extend the sandbox to handle them.

References

Forišek, M. (2006). Security of programming contest systems. In: Dagiene, V., Mittermeir, R. (Eds.), *Information Technologies at School*, 553–563.

Jana, S. *et al.* (2011). TxBox: building secure, efficient sandboxes with system transactions. In: *Proceedings of the 32nd IEEE Symposium on Security and Privacy*, 329–344, Austin, TX, USA.

Kerrisk, M. (2010). *The Linux Programming Interface*. No Starch Press.

Kerrisk, M. *et al.* (2012). *The Linux Man-Pages Project*. Available on-line at `http://www.kernel.org/doc/man-pages/`.

Kolstad, R. (2009). Infrastructure for contest task development. *Olympiads in Informatics*, 3, 38–59.

Mareš, M. (2007). Perspectives on grading systems. *Olympiads in Informatics*, 1, 124–130.

Mareš, M. (2009). Moe — design of a modular grading system. *Olympiads in Informatics*, 3, 60–66.

Mareš, M. (2011). Fairness of time constraints. *Olympiads in Informatics*, 5, 92–102.

Merry, B. (2009). Using a Linux security module for contest security. *Olympiads in Informatics*, 3, 67–73.

Merry, B. (2010). Performance analysis of sandboxes for reactive tasks. *Olympiads in Informatics*, 4, 87–94.

Wahbe, R. *et al.* (1993). Efficient software-based fault isolation. In: *Proceedings of the 14th Symposium on Operating Systems Principles*. 203–216, New York, NY, USA.

Yee, B. *et al.* (2009). Native client: a sandbox for portable, untrusted x86 native code. *IEEE Symposium on Security and Privacy*, Oakland, USA.

M. Mareš is as an assistant professor at the Department of Applied Mathematics of Faculty of Mathematics and Physics of the Charles University in Prague, organizer of several Czech programming contests, member of the IOI Scientific Committee, and a Linux hacker.

B. Blackham is a PhD student with the Software Systems Research Group at NICTA and the School of Computer Science & Engineering at the University of New South Wales in Sydney, Australia. He was formerly the team leader of the Australian IOI team and has been involved with training the Australian team for 10 years.