Secure, User-level Resource-constrained Sandboxing

Fangzhe Chang, Ayal Itzkovitz, and Vijay Karamcheti
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
{*fangzhe, ayali, vijayk*}*@cs.nyu.edu*

Abstract

The popularity of mobile and networked applications has resulted in an increasing demand for execution "sandboxes"—environments that impose irrevocable qualitative and quantitative restrictions on resource usage. Existing approaches either verify application compliance to restrictions at start time (e.g., using certified code or language-based protection) or enforce it at run time (e.g., using kernel support, binary modification, or active interception of the application's interactions with the operating system). However, their general applicability is constrained by the fact that they are either too heavyweight and inflexible, or are limited in the kinds of sandboxing restrictions and applications they can handle.

This paper presents a secure user-level sandboxing approach for enforcing both qualitative and quantitative restrictions on resource usage of applications in distributed systems. Our approach actively monitors an application's interactions with the underlying system, proactively controlling it as desired to enforce the desired behavior. Our approach leverages a core set of user-level mechanisms that are available in most modern operating systems: fine-grained timers, monitoring infrastructure (e.g., the /proc filesystem), debugger processes, priority-based scheduling, and page-based memory protection. We describe implementations of a sandbox that imposes quantitative restrictions on CPU, memory, and network usage on two commodity operating systems: Windows NT and Linux. Our results show that application usage of resources can be restricted to within 3% of desired limits with minimal run-time overhead.

1 Introduction

The increasing availability of network-based services and the growing popularity of mobile computing has resulted in a situation where current-day distributed applications are often built up (possibly dynamically) from components originating from a variety of sources. Since an end-user cannot be expected to trust all of these sources, there is an increasing demand for execution "sandboxes"—environments that impose irrevocable qualitative and quantitative restrictions on resource usage. For example, the execution environment can ensure that the application component can only access certain portions of the file system (e.g., `/tmp`), or that running the component would consume no more than 20% of the CPU share. These restrictions isolate the behavior of other activities on the system from a potentially malicious component, and are a precondition for the wider deployment of distributed component-based applications.

Existing approaches for enforcing qualitative and quantitative restrictions on resource usage can be classified into two broad classes: those that verify application compliance to restrictions at start time and those that enforce it at run time. Examples of the first class include approaches that rely on certified code [16, 17] or language-based protection [1, 3]. These approaches have the limitation of lacking generality (because of reliance on specific programming languages and compilers) and are typically unable to enforce quantitative restrictions. Examples of the second class include approaches that rely on kernel support [13, 15],

binary modification [18], or active interception of the application's interactions with the operating system (OS) [2, 8, 9] for isolating resource usage. The kernel approaches are general-purpose but require extensive modifications to OS structure and lack flexibility with respect to what restrictions are imposed. The remainder of the approaches rely on deciding for each application interaction with the underlying system whether or not to permit this interaction to proceed; consequently, they provide some flexibility with respect to enforcing qualitative restrictions but are unable to handle most kinds of quantitative restrictions (particularly since usage of some resources, e.g. the CPU, does not require explicit application requests).

This paper presents a secure, user-level sandboxing approach for enforcing both qualitative and quantitative restrictions on resource usage of general-purpose applications in distributed systems. The qualitative aspect is similar to previous systems referred to above; what is novel however, is the ability of our approach to enforce quantitative constraints. Our approach actively *monitors* an application's interactions with the underlying system, proactively *controlling* it as desired to enforce the desired behavior. The security of the approach stems from implementation mechanisms that prevent a malicious application from being able to undo the monitoring and control. Our general strategy recognizes that application access to system resources can be modeled as a sequence of requests (either implicit such as for a physical memory page, or explicit such as for a disk operation) spread out over time. This observation provides two alternatives for constraining resource utilization: either control the resources available to the application at the point of the request or (in the case of resources with rate constraints such as CPU and memory) control the time interval between resource requests. In both cases and for all kinds of resources, the specific control is influenced by the extent to which the application has exceeded or fallen behind a *progress metric*. The latter represents an estimate of the resource consumption of the application program.

Although the high-level strategy is relatively straightforward, the primary challenge lies in accurately estimating the progress metric and effecting necessary control on resource requests with minimal overhead. It might appear that appropriate monitoring and control would require extensive kernel involvement, restricting their applicability. Fortunately, most modern OSes provide a core set of user-level mechanisms that can be used to construct the required support. Support for *fine-grained timers* and *monitoring infrastructures* such as the UNIX `/proc` filesystem and the Windows NT Performance Counters provide needed information for building accurate progress models. Similarly, fine-grained control can be effected using standard mechanisms for *debugger processes*, *priority-based scheduling*, and *page-based memory protection*.

We describe implementations of a sandbox that imposes quantitative restrictions on usage of three representative resources—CPU, memory, and network—on two commodity operating systems: Windows NT and Linux. The two implementations utilize the same high-level strategy, but rely on platform-specific monitoring and control mechanisms. A detailed evaluation shows that both sandbox implementations are able to restrict resource usage of unmodified applications to within 3% of the prescribed limits with minimal run-time overhead. We also present a synthetic application that demonstrates the flexibility advantages of user-level sandboxing: in this case, our approach permits application-specific control over resource usage at granularities smaller than those controllable using kernel-level mechanisms.

The specific contributions of this paper include:

- A general user-level strategy for exploiting widely available OS features to impose quantitative restrictions on resource usage, and for securing the sandbox despite its user-level implementation.
- Two concrete implementations of this strategy on Windows NT and Linux operating systems using as example three representative resource types (CPU, memory, and network).
- An evaluation of the overheads and flexibility of the user-level sandboxing approach.

The rest of this paper is organized as follows. Section 2 provides background and discusses related work. Section 3 presents the overall sandboxing strategy and discusses its application for three example resource types: CPU, memory, and network. Section 4 shows how to secure this strategy against malicious

applications. Concrete implementations of the sandbox on Windows NT and Linux OSes are presented and evaluated in Sections 5 and 6. Section 7 highlights the flexibility advantages of user-level sandboxing, and we conclude in Section 8.

2 Background and Related Work

The problem of ensuring that untrusted application components in a distributed system do not violate certain qualitative and quantitative restrictions on resource usage has recently attracted a lot of attention. Existing approaches can be classified into two broad classes: those that verify application compliance to restrictions at start time and those that enforce it at run time.

2.1 Enforcing compliance at start time

Several mechanisms can be used to verify, prior to starting execution of a component, that the latter satisfies the restrictions imposed upon it by the environment. These mechanisms include:

- relying on a certificate authority (e.g., VeriSign [17]) to attest to the fact that the application component satisfies the desired properties,
- using language-based protection techniques such as type safety [3] or static bytecode verification [14] to verify that the program does not misbehave, and
- using techniques such as proof-carrying code [16] that permit a verifier on the client machine to confirm that the application component satisfies certain safety properties.

The above approaches are very effective at ensuring that the application does not violate qualitative restrictions (e.g., that certain types of resources are not accessed from specific code modules), but are typically unable to enforce quantitative restrictions. This is because the former can be expressed as safety properties, easier to verify statically. In addition, some of these approaches lack generality because of reliance on specific programming languages and compilers.

2.2 Enforcing compliance at run time

Approaches for enforcing run-time compliance of application behavior fall into two sub-categories:

Kernel-level mechanisms such as CPU reservations [13, 15] and fair-share queueing of CPU [10, 19] and network resources [6, 7, 20] have been employed, primarily in the context of real-time operating systems, to enforce both qualitative and quantitative restrictions. Actually, such support provides a stronger guarantee of a certain level of resource allocation over a time window. Some restricted versions of such mechanisms are also likely to be available in the form of job control mechanisms in Windows 2000 (announced).

Although such approaches are general-purpose, they require extensive modifications to OS structure that limits their applicability to commodity OSes. Additionally, the policy space of restrictions that can be imposed is inflexible, being limited to the few options that have been designed into the kernel.

Code transformation techniques provide a user-level approach for imposing restrictions on resource usage. These techniques, which include binary modification approaches such as software fault-isolation [18] and API interception approaches such as Janus [9], Mediating Connectors [2], and Naccio [8], all rely on monitoring an application's interactions with the underlying OS. These techniques leverage OS mechanisms such as system-call interception by a debugger process [9], or application structuring mechanisms such as DLL import-address-table rewriting [2, 12] to execute some checking code whenever the application

interacts with the OS. This code decides, for relevant interactions, whether to allow or deny the interaction from proceeding.[1]

Consequently, such approaches provide some flexibility with respect to enforcing qualitative restrictions (e.g., that all memory loads/stores are to certain reserved portions of the application address space), but are unable to handle most kinds of quantitative restrictions. The latter is particularly true because usage of some resources, e.g. the CPU, does not involve explicit application requests.

Our approach uses the same underlying mechanisms as the code transformation techniques described above, but differs in its ability to also enforce quantitative restrictions over resource usage. To achieve this, as we describe in the next section, it builds upon core monitoring and control mechanisms that are a feature of most modern OSes.

3 Capacity Sandboxing: Enforcing Quantitative Restrictions

Although our sandboxing approach enforces both qualitative and quantitative restrictions on resource usage, we restrict our attention in the remainder of the paper to quantitative (capacity) constraints. For the former, we rely on the relatively well-understood code transformation approach that has been described earlier. The basic idea is that access control and other qualitative constraints can be enforced by intercepting application interactions with the underlying OS and appropriately modifying them to ensure compliance with the desired security property. For example, an application program can be prevented from accessing files in directories other than a designated one by ensuring that the `fopen` API call is intercepted and modified to reflect a file name and path relative to this designated directory.

Enforcing quantitative (or capacity) constraints (e.g., a client program does not use more than 20 MB of RAM) is more involved. This is because several system resources such as the CPU and memory can be accessed without going through a high-level API call that can be intercepted. Moreover, individual API calls may not provide useful information to determine whether or not the client program has exceeded its capacity constraints and what needs to be done to rectify that. In the rest of this section, we first describe the general architecture for our capacity sandbox, and then exemplify its usage.

3.1 General Architecture

To enforce quantitative restrictions on usage for resources of different types, our general strategy relies on the recognition that application access to system resources can be modeled as a sequence of requests (either implicit such as for a physical memory page, or explicit such as for a disk operation) spread out over time. This observation provides two alternatives for constraining resource utilization: either control the resources available to the application at the point of the request, or (in the case of resources with rate constraints such as CPU, network, and disk) control the time interval between resource requests. In both cases and for all kinds of resources, the specific control is influenced by the extent to which the application has exceeded or fallen behind a *progress metric.* The latter represents an estimate of the consumption of a particular kind of resource over a time slot.

This general strategy is realized by relying upon techniques for *instrumenting the application, monitoring its progress,* and as necessary, *controlling its progress* (see Figure 1). Instrumenting allows us to modify the application code so that control over the application is possible. Monitoring enables us to be aware of the current state of the application and its utilization of various resources. Finally, controlling the application progress is the proactive mechanism for enforcing quantitative restrictions on resource usage. All three sets

[1] Or, in some cases (e.g., Janus [9]), to modify the request into a compliant form prior to allowing it to proceed.

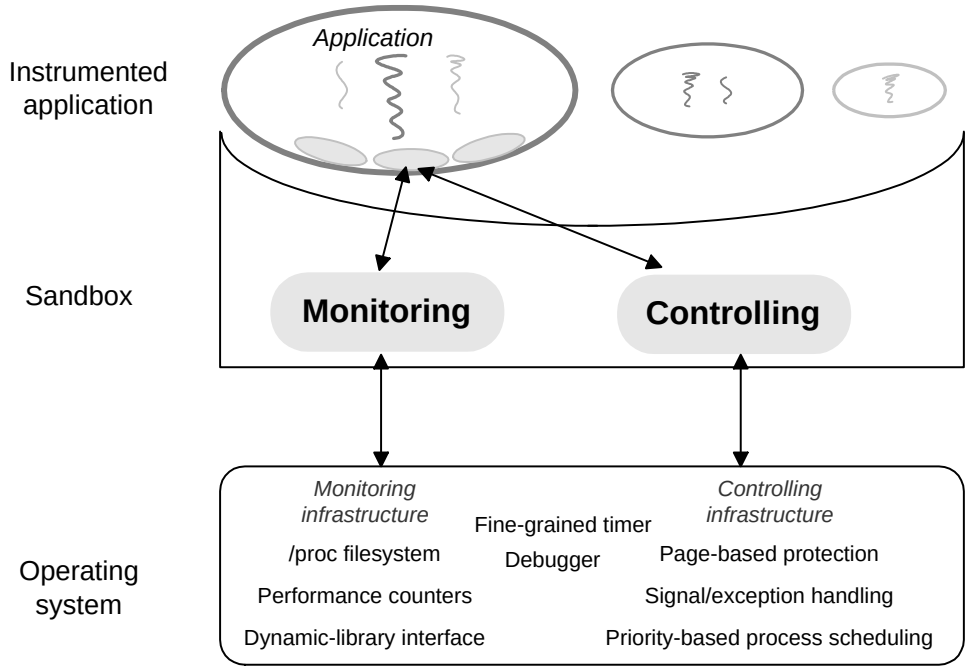


Figure 1: General sandboxing strategy that imposes quantitative restrictions on resource usage.

of techniques leverage a core set of user-level mechanisms that are provided by most modern OSes. In the following discussion, these mechanisms are shown underlined and in bold.

Instrumenting the application Instrumentation refers to the technique of modifying the application code on the fly, without having to recompile or relink the application. It leverages the fact that modern OSes provide a significant portion of their functionality as shared libraries whose interfaces are well defined. The application interactions at this interface can be intercepted using OS-specific techniques. The latter can range from something as simple as library preloading on Linux to more OS-supported mechanisms such as the `ptrace` facility on several Unix-based OSes that permits a **debugger process** to be informed whenever the process being debugged makes a system call. Similar mechanisms exist on the Windows NT OS ranging from rewrites to the process DLL import address table [2] to more extensive modifications to the process address space [12] using a debugger process capabilities. Interception of the application interactions permits the injection of code, which can monitor and control application behavior (in particular, resource utilization).

Our approach relies on instrumentation to enable a distinguished process, the *monitor,* to load the application components that need to run within a sandbox and be able to monitor and control the behavior of these processes by injecting required functionality into their address space. The monitor also maps a shared memory segment for coordination with the injected code.

Monitoring progress Monitoring, as well as accounting, of application behavior is well-supported on modern OSes. Extensive information about process usage of various resources (e.g., CPU, physical memory) can be obtained through various **monitoring infrastructures**. The latter range from special system calls, to consulting the registry of the dynamic performance data (on Windows NT), to a file-system based interface to this data (e.g., the `/proc` mechanisms on Unix systems). Although our strategy relies on the availability of such mechanisms to obtain progress data, in some cases this data is not very up-to-date or incurs high overheads for its update. In those situations (e.g., deciding whether or not a process is waiting for a system

event), the debugger interface permits faster access to the desired information.

An additional issue is that most of the mechanisms referred to above only provide accumulated information (since the start of the process). So, this information is periodically sampled with the sampling period automatically adapted to the rate at which the application consumes requests. For example, usage of network resources can be monitored/updated whenever the application executes a networking API call. For resources that are not accessed through explicit API calls (e.g., CPU and memory), the monitoring information can be periodically updated using support for **fine-grained timers**. The latter allow association of periodic activity at a granularity of sub-10ms on current-day OSes.

Controlling progress Upon detecting that the progress metric for an application component has exceeded or fallen behind a resource threshold, our strategy controls application behavior to enforce compliance. The actual mechanism used for controlling progress depends upon whether or not application use of a resource involves an explicit API call. If it does (e.g., for use of network or disk resources), appropriate control can be achieved by limiting the resources available at the point of request or by varying the time interval between resource requests. The latter relies on **fine-grained process sleep** operations on modern OSes.

When resource usage is implicit, our strategy relies on two sets of mechanisms that: (1) (for CPU) control how frequently the application is scheduled (thereby effecting a delay between resource usage requests) by leveraging support for **priority-based process scheduling**, and (2) (for memory) return allocated resources back to the OS using a resource-specific protocol. For instance, memory resources can be relinquished by setting page protection bits to `NoAccess` (on NT), or unmapping appropriate portions of the virtual address space. The controlling code must also ensure that the application continues to function correctly despite this loss of resources. In the case of memory resources, the controller takes advantage of OS support for **page-based protection** and **user-level protection fault handlers** to invoke this functionality.

3.2 Constraining Usage for Different Resource Types

We consider three representative resources—CPU, memory, and network—to illustrate the above strategy. Implementation details on Windows NT and Linux are deferred to Sections 5 and 6.

CPU Resources Here, the quantitative restriction is to ensure that the application receives a stable, predictable processor share. From the application's perspective, it should appear as if it were executing on a virtual processor of a certain speed.

Constraining CPU usage of an application utilizes the general strategy described earlier. The monitor process periodically samples the underlying performance monitoring infrastructure to estimate a progress metric. In this case, progress can be defined as the portion of its CPU requirement that has been satisfied over a period of time. This metric can be calculated as the ratio of the allocated CPU time to the total time this application has been ready for execution in this period. However, although most OSes provide the former information, they do not yield much information on the latter. This is because few OS monitoring infrastructures distinguish (in what gets recorded) between time periods where the process is waiting for a system event and where it is ready waiting for another process to yield the CPU. To model the virtual processor behavior of an application with wait times (see Figure 2 for a depiction of the desired behavior), we estimate the total time the application is in a wait state using a heuristic. The heuristic periodically checks the process state either by querying the monitoring infrastructure or by inspecting the process stacks, and assumes that the process has been in the same state for the entire time since the previous check.

The actual CPU share allocated to the application is controlled by periodically determining whether the granted CPU share exceeds or falls behind the desired threshold. The guiding principle is that if other applications take up excessive CPU at the expense of the sandboxed application, the monitor compensates by

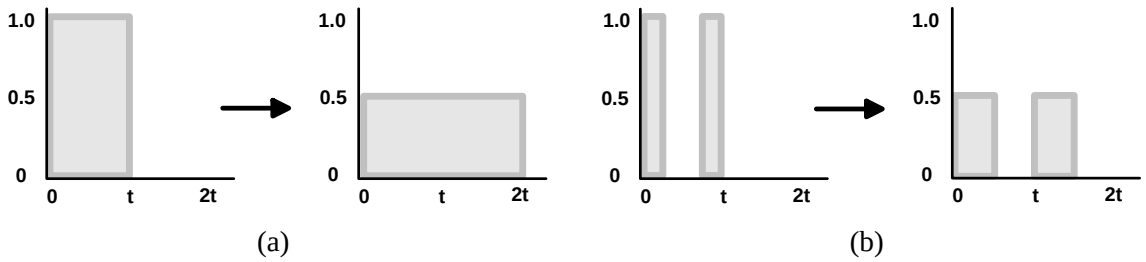


Figure 2: Desired effects on application execution time under a resource-constrained sandbox that limits CPU share (50% in this case) when the application contains (a) no wait states, and (b) with wait states.

giving the application a higher share of the CPU than what has been requested. However, if the application's CPU usage exceeds the requested processor share, the monitor would reduce the CPU quantum it gets for a while, until the average utilization drops down to the requested level. The scope of these adjustments (i.e., lifetime of the application) needs to be larger than the time period between sampling points where the progress metric is recomputed.

Memory Resources The quantitative restriction of interest here is the amount of physical memory an application can use. The sandbox would ensure that physical memory allocated to the application does not exceed a prescribed threshold. Monitoring the amount of physical memory allocated to an application is straightforward. The monitoring infrastructure on all modern OSes provides this information in the form of the process resident set size.

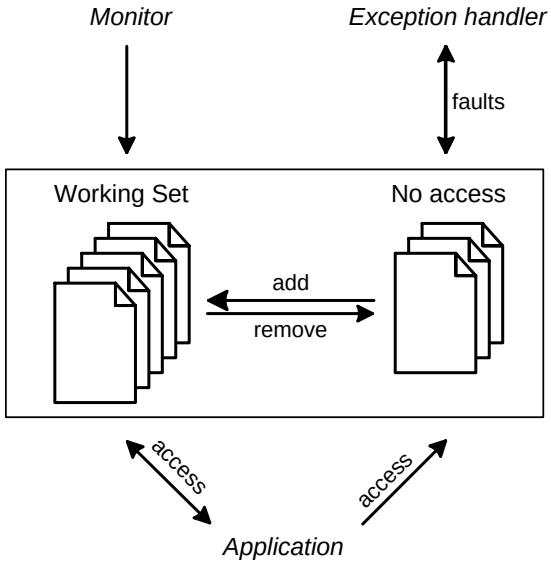


Figure 3: A general user-level strategy for controlling application physical memory usage.

However, it is more involved to control the application behavior in case the OS allocates more physical pages than the threshold. The problem is that these resources are allocated implicitly subject to the OS memory management policies. The basic idea is to have the monitor act as a user-level pager on top of the OS-level pager, relying on an OS-specific protocol for voluntarily relinquishing the surplus physical mem-

ory pages allocated to the application (see Figure 3). Also, unlike the CPU case where periodic monitoring and control of application progress is required, here the monitoring and control can adapt itself to application behavior. The latter is required only if the application physical memory usage exceeds the prescribed threshold, which in turn can be detected by exploiting OS support for user-level protection fault handlers.

Network Resources Here, the quantitative restriction refers to the sending or receiving bandwidth available to the application on its network connections. Unlike CPU and memory resources, application usage of network resources involves an explicit API request. This permits the monitoring code injected into the application to keep track of the amount of data sent over a time window and estimate the bandwidth available to the application. Control is equally straightforward: if the application is seen to exceed its bandwidth threshold, it can be made compliant by just stretching out the data transmission or reception over a longer time period (e.g., by using fine-grained sleeps).

Although the above description works well for controlling bandwidth into and out of an end-point,[2] a different monitoring and controlling approach is required when the sandbox environment must be extended to control network bandwidth for connections between multiple hosts. In this case, simple modeling of the bandwidth results in a situation where bursts in network traffic are not modeled accurately (since these get smoothed out at each end-point). In order to control the network bandwidth B, we need to know when data is sent and its size. The idea is as follows. When the application performs a send(), we inspect the current network usage and decide the emulated amount of bandwidth this send() can get. In case there are no other "pending" send()s, this value is B, but if there are other preceding send()s that still take up some bandwidth, a lower rate B' might be given so that the overall sending bandwidth is B. Knowledge of the emulated bandwidth B' allows computation of an appropriate delay parameter that models the message send being stretched out by the right amount. After this delay, the entire message is sent out in its original form. With this scheme, the receiver will observe the same bandwidth as the sender, which might be sufficient in cases where there is low contention in the communication pattern between hosts.

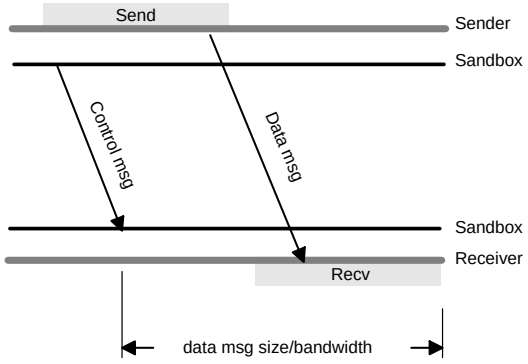


Figure 4: A general strategy for imposing global network bandwidth control for distributed applications.

In the presence of contention, limiting only the sending bandwidth is not enough. The receiver might accept connections from multiple hosts, thus it is necessary to constrain the incoming message rate according to the receiving bandwidth restrictions, as in a real network. Limiting the receiving bandwidth requires computation of bandwidth from the start of the send operation to the end of the receive operation. However, as stated above, there is no way for the monitoring/controlling code on the receiver to know when a particular message was sent (since there is no way to factor out contention in the network). The ideal solution would be

[2]For clarity of description, we restrict our attention in this paper to synchronous communication operations and also assume that the data transmission rate in the network is not the bottleneck. The approach needs to be refined slightly to handle situations where communication operations are asynchronous.

to timestamp each message with a global clock and have the receiving code use this timestamp to compute emulated bandwidth. However, given the infeasibility and high overheads of building synchronized clocks in a distributed system, we approximate its behavior by having the monitoring code periodically exchange control messages (see Figure 4). Each control message defines the extent of a virtual time window on the originating host, and indicates how much data has been sent to the peer in the previous time window. The receiving node records the local time of reception of a control message and uses it to infer whether or not data messages it has received need to be delayed to emulate the desired bandwidth constraints. Note that this scheme is robust to control messages arriving later than some of the data messages whose information they carry: this would be taken to imply that the receive has in fact been issued in time and any required delays are just added on to future message receptions.

In Sections 5 and 6, we describe concrete implementations and performance of the strategy described here on two commodity OSes: Windows NT and Linux.

4 Securing the Sandbox

The sandboxing strategy described in the previous section is secure if the monitoring and controlling functionality is embedded in the debugger process that loads the client program being sandboxed. In this case, traditional OS process protection mechanisms and the asymmetric relationship between the debugger and debuggee processes are sufficient to ensure that the client program cannot undo the qualitative and quantitative restrictions imposed upon it.

However, on some operating systems and more generally for performance reasons (to minimize sandboxing overheads) we might want to inject some of the monitoring and controlling functionality into the client process itself. This leads to a potential vulnerability: since this functionality is part of the user address space, a malicious client program might be able to undo the restrictions. We have developed a strategy for securing the sandbox despite its user-level nature. For space reasons, we only sketch the overall scheme here. A complete description and evaluation of the scheme is the subject of a forthcoming paper [4].

Our user-level sandboxing strategy guarantees irrevocability by exploiting the observation that the sandboxing code gets a chance to initialize itself before the client program. Therefore, it is possible for the sandboxing code to first modify the code images in the client program's virtual address space as appropriate and then leave it in a state such that *neither the client program nor the sandboxing code itself* can undo the effects of this modification.

In more detail, there are two main threats that the sandboxing code must counter (see Figure 5):

1. Prevent the client program from undoing any changes to the code images in its virtual address space.
2. Prevent the client program from bypassing the intercepted path for invoking operating system services.

Preventing modification to sandboxing code To ensure that the first threat does not succeed, the sandboxing code can write the modified code images, protect the pages containing these modified images to prevent further modification, and finally prevent the client program from resetting these protection bits. The latter introduces a bootstrapping problem because the sandboxing program must initially be allowed to reset protection bits. This can be resolved by modifying the API call responsible for changing page protections (`mprotect` in Unix) in a way such that after desired modifications have been effected, this API call atomically switches into a mode where further illegal page protection changes are disallowed. Our implementation relies on self-modifying code to achieve this.

Preventing bypassing of sandboxing code To prevent the second threat from succeeding, we need to ensure that the client program (a) does not invoke the same functionality when loaded into a different portion

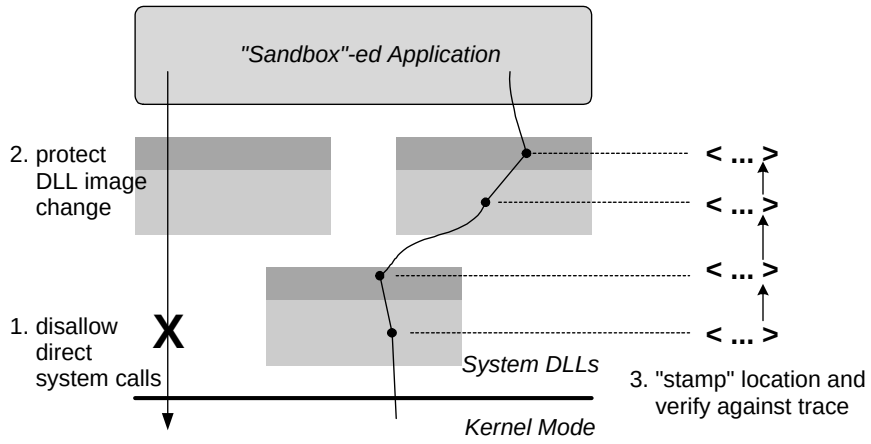


Figure 5: User-level sandboxing is secured by disallowing direct entry into the kernel, protecting DLL images from further modification, and tracing the execution path to prevent bypass of sandboxing code.

of its virtual address space, (b) does not directly enter kernel mode by executing a trap instruction or other means, and (c) does not directly invoke the internal API function that is being intercepted.

Case (a) can be handled by treating intercepted libraries as part of the trusted code base and not permitting them to be reloaded. This does not prevent the client program from embedding the functionality of these libraries elsewhere and then invoking it, but this possibility is handled the same way as case (b). For this case, we need to ensure that the client program cannot enter kernel mode using either a trap instruction or other means (such as a LPC call in Windows NT). This is ensured by scanning the client program (and by extension, any libraries loaded by it) for offensive instructions and preventing cross-protection domain jumps using software fault-isolation [18] techniques. This still leaves open the possibility that the client program assembles an impermissible instruction sequence in its data segment and then calls/jumps to it.[3] We prevent this case from happening by using software fault-isolation techniques to ensure that each call/jump in the client program is to a "verified" code segment.

Case (c) requires ensuring that all calls to API functions that have been intercepted first go through the intercepted code. The general problem represented by this statement is one of ensuring that process execution follows a specified path, say denoted by a sequence of instruction addresses. The strategy we employ to verify this consists of "stamping" the current location in the path using a spoof-free technique. The "stamping" operation verifies that the previous stamp corresponds to the previous location in the path and updates the stamp information. The spoof-free technique used for the stamping operation relies on the inability of the client program to spoof its current instruction pointer register on a call: this serves as an indication of whether or not the stamping operation originated at a valid place in the program. The sandboxing code can then check the call stack against this stamp trace to verify that the code fragments implementing the sandbox have not been bypassed in the execution path.

The strategy we have outlined above is similar in spirit to the low-level safety mechanisms employed in the Naccio [8] system. One significant difference is our use of the stamping strategy, which permits the original (unintercepted) functions to continue being a part of the address space. Naccio on the other hand, assumes that all system libraries have been instrumented prior to their being loaded.

[3] Ideally, this could be handled by preventing code in pages not marked executable from being executed. Unfortunately, few current-day processors enforce this distinction.

5 Implementing the Sandbox on Windows NT

The implementation of the resource-constrained sandbox on Windows NT follows the general strategy described in Section 3. This section discusses NT-specific issues and demonstrates the control of CPU, memory, and network resources with experiments. The implementation and performance results below refer to NT Version 4, service patch 5.

5.1 Constraining use of CPU resources

Monitoring progress The CPU monitor is attached as a callback routine of the fine-grained *multimedia timers*, and can be triggered every 10ms with high accuracy using a technique introduced in [11]. Note that the scheduling quantum on NT is at least 20ms for the workstation installation and 120 ms for the server installation. As described in Section 3.2, the monitor estimates process wait time within a time window by checking the process state and accumulating the time slots at which the process is found waiting. Although NT allows examining process state via its performance counters infrastructure, this incurs high overhead (on the order of milliseconds). Instead, we employ a heuristic that infers process state based on the thread contexts. A thread can be in a wait state only when it is executing a function inside the kernel. Recognizing that if the thread is not blocked it is unlikely to stay at the same place in kernel code, the heuristic checks the instruction pointer register to see whether a trap instruction (int 2Eh) has just been executed, and whether any general registers have changed since the last check. If the same context is seen, it regards the thread as being in a wait state, with the process regarded as waiting if all of its threads are waiting.

Controlling progress Based on the progress metric, the controlling code decides whether or not to schedule the process in the next time slot. Although this decision could be implemented using NT support for suspending and resuming threads, the latter incurs high overheads. Consequently, we adopt a different strategy that relies on fine-grained adjustment of application process priorities to achieve the same result.

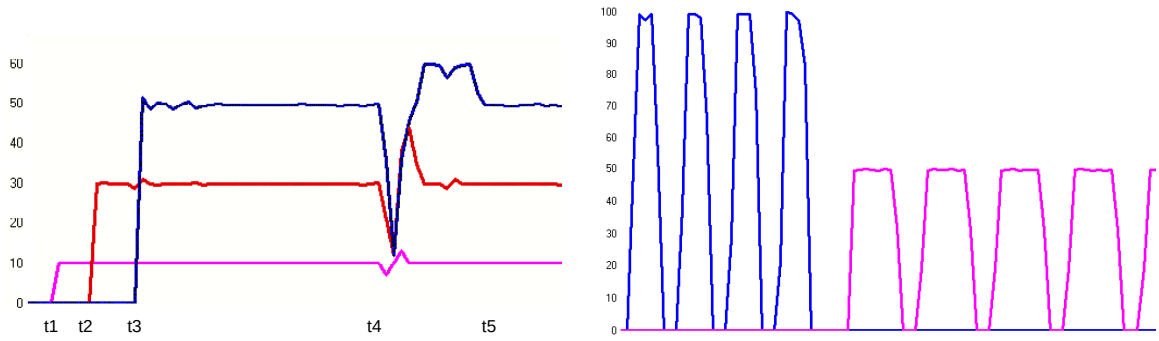
Priority level	CPU available	CPU not available
4	`Monitor`	`Monitor`
3	**Application**	
2	`Hog`	`Hog`
1		**Application**

Figure 6: Controlling application CPU availability by changing priorities.

Our approach requires four priority classes (see Figure 6), two of which encode whether CPU resource are available or unavailable to the application. The monitor runs at the highest priority (level 4), and a special compute-bound "hog" process runs at priority level 2. An application process not making sufficient progress is boosted to priority level 3, where it preempts the hog process and occupies the CPU. A process that has exceeded its share is lowered to priority level 1, allowing other processes (possibly running within their own sandboxes) or in their absence, the hog, to use the CPU. Note that this scheme allows multiple sandboxes to coexist on the same host.

Effectiveness of the sandbox Our experiments show that this implementation enables stable control of CPU resources in 1% to 97% range. When the requested share is above 97%, the measured allocation

includes perturbations from background load (the performance monitor, system processes, and the sandboxing code). The interference from sandboxing code includes the monitor overhead (negligible) and bursty allocation of resources to the hog process over long runs (this is an NT feature for avoiding starvation).



(a) (b)

Figure 7: (a) Weighted CPU sharing for multiple applications. (b) Constraining CPU share for applications with wait states.

Figure 7(a) is a snapshot of the performance monitor display showing three sandboxed applications running on the same host. They start at times $t1$, $t2$, and $t3$, requesting 10%, 30%, and 50% of the CPU share, respectively. With the total CPU load at 90%, all three applications receive a steady CPU share until time $t4$, when we deliberately perturb the allocation by dragging a window. This causes the total available CPU to decrease drastically (because of the kernel activity), and a sharp decrease in the immediate CPU shares available to each application. However, this drop is compensated with additional CPU resources once the system reacquires CPU resources (end of window movement). These results indicates that sandbox can support accurate and stable CPU sharing with resilient compensation.

Figure 7(b) shows the execution of an application that sleeps periodically, without sandboxing (left) and with a sandboxed CPU share of 50% (right). The working time with the sandbox is twice the amount on the left, corresponding to the halved CPU resource. More importantly, the sleep (wait) time is kept the same, consistent with Figure 2 and verifying the effectiveness of our state-checking heuristic.

5.2 Constraining use of memory resources

Monitoring progress An API call, `GetProcessMemoryInfo`, provides information about the resident memory of a process. Unlike the CPU case, the sampling of this information can be adapted to the rate at which the application consumes memory resources. To estimate the latter, we integrate the sampling with the controlling scheme described below.

Controlling progress As described in Section 3.2, controlling progress of memory resources requires the sandboxing code to relinquish surplus memory pages to the OS. To do this, we rely on a convention in NT: pages whose protection attributes are marked `NoAccess` are collected by the swapper.

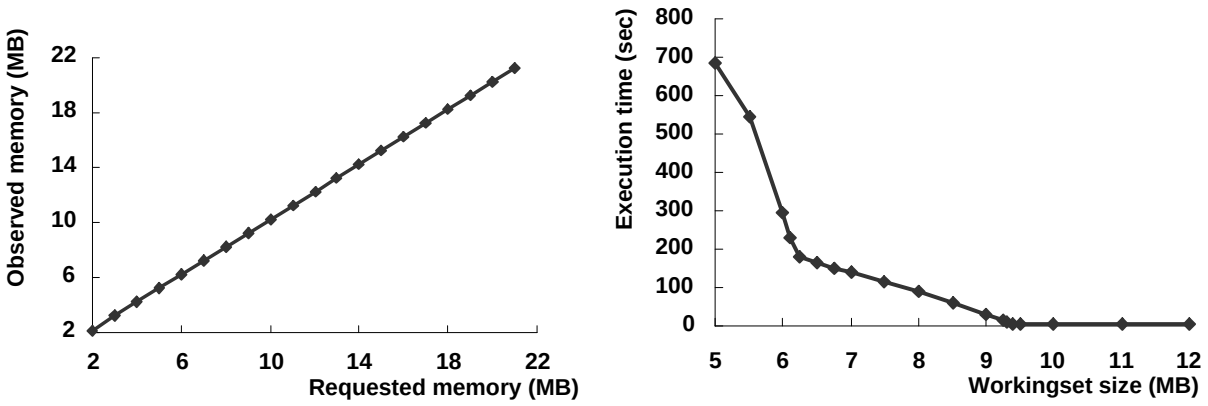
The same core OS mechanism, user-level protection fault handlers, is used to decide (a) *when* a page must be relinquished, and (b) *which* page must this be. Our scheme instruments the memory allocation APIs (e.g., `VirtualAlloc` and `HeapAlloc`) to build up its own representation of the process working set. When the allocated pages exceed the desired working set size, extra pages are marked `NoAccess`. When such a page is accessed, a protection fault is triggered, the sandbox catches this fault and changes page

protection to `ReadWrite`. Note that this might enlarge the working set of the process, in which case a FIFO policy is used to evict a page from the (sandbox-maintained view of the) working set. The protection fault handler also provides a natural place for sampling the actual working set size.

A few additional points need clarification. The implementation is simplified by not evicting pages containing the executable code and execution stack, so this limits the least amount of memory that can be constrained. Eviction at the sandbox level may or may not cause the page to be written to disk although these pages are excluded from the process working set; when the system has large amounts of free memory, NT maintains some pages in a transition state delaying writing them to disk. Note also that with our design, if the application is running within its memory limits, it will not suffer from any runtime overhead. Beyond that point, the overheads are a function of process virtual memory locality behavior.

Effectiveness of the sandbox Our experiments show that, on a 450 Pentium II machine with 128MB memory, this sandbox implementation can effectively control actual physical memory usage for arbitrary applications from 1.5MB up to around 100MB.

Figure 8(a) shows the requested and measured physical memory allocations for an application that has an initial working set size of 1.5MB and allocates an additional 20MB of memory. The sandbox is configured to limit available memory to various sizes ranging from 2MB to 21MB. As the figure shows, the measured memory allocation of the application (read from the NT Performance Monitor) is virtually identical to what was requested.



(a) (b)

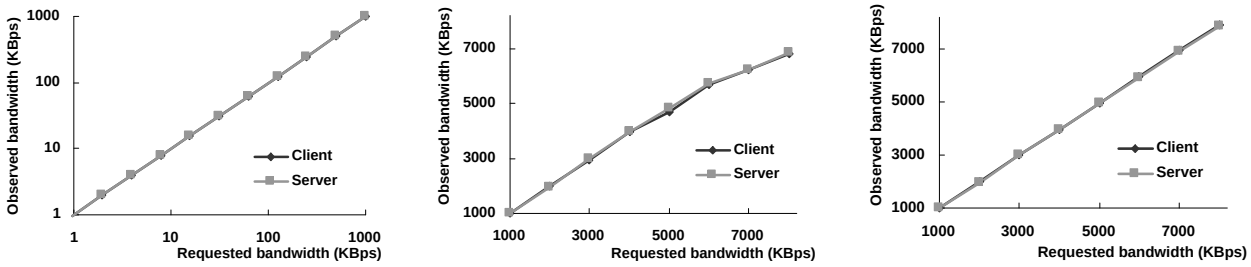
Figure 8: (a) Controlling the amount of physical memory utilized by an application. (b) Execution time as size of working set varies.

Figure 8(b) demonstrates the impact of the memory sandbox on application execution time. The application in question has a memory access pattern that produces page faults linearly proportional to the non-resident portion of its data set. In this case, the application starts off with a working set size of 1.5MB and allocates an additional 8MB. The sandbox enforces physical memory constraints between 5MB and 12MB. As the figure shows, the execution time behavior of the application can be divided into three regions with different slopes. When the memory constraint is more than 9.5MB, all of the accessed data can be loaded into physical memory and there are no page faults. When the memory constraint is below 9.5MB, total execution time increases linearly as the non-resident size increases, until the constraints reaches 6.25MB. In this region, page faults occur as expected but the process pages are not written to disk. When available memory is below 6.25MB, we observe heavy disk activity. In this segment, the execution time also varies approximately linearly, with the slope determined by disk access characteristics.

5.3 Constraining available network bandwidth

Monitoring and controlling progress As described in Section 3.2, we intercept socket APIs (*accept, connect, send,* and *recv*) to monitor and control available network bandwidth. We implemented both local and global network control mechanisms, but report here only the performance of the former.

Effectiveness of the sandbox The effectiveness of the sandbox was evaluated on a pair of Pentium II (450 MHz) machines connected to a 10/100 auto-sensing Fast Ethernet hub. The application consisted of a server and one or more clients in a simple ping-pong communication pattern. The peak bandwidth measured without the sandbox was 9.7MBps, whereas the sandbox permitted effective control of bandwidth over the range 1 Bps to 8.8 MBps.



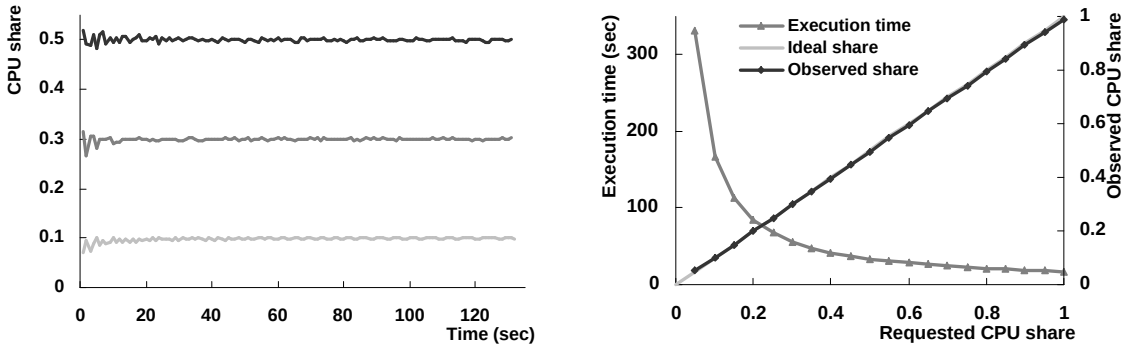
(a) (b) (c)

Figure 9: Measured bandwidth (a) (for 1 KB messages) as requested bandwidth varies from 1 KBps to 1000 KBps, (b) (for 1 KB messages) as requested bandwidth varies from 1000 KBps to 8000 KBps, (c) (for 10 KB messages) as requested bandwidth varies from 1000 KBps to 8000 KBps.

Figure 9(a) shows the measured bandwidth (at both server and client) as the sandbox constrains network bandwidth from 1 KBps to 1000 KBps. In the figure, the server and client bandwidth lines are virtually indistinguishable from each other and within 1% of the requested limit. Figure 9(b) shows the same measurement when bandwidth constraints are applied in the range 1000 KBps to 8000 KBps. When the requested bandwidth is below 4000 KBps, the sandbox enforces the request with an error of at most 2%. However, error grows for higher bandwidths. For example, at 8000 KBps, the sandbox can sustain a bandwidth at best 16% lower than requested. This is not surprising, since each 1 KB message incurs an overhead of 125 μs at 8000 KBps, and the monitoring and controlling code adds significant overhead. Figure 9(c), using 10 KB messages, shows that the sandbox can be used to accurately sustain higher requested bandwidth as long as the message size scales proportionally. Here, the error between measured and requested bandwidths is less than 2%.

6 Implementing the Sandbox on Linux

Linux provides support very similar to Windows NT for instrumenting application binaries, and monitoring and controlling resource consumption. For instance, library functions such as the sockets and memory allocation APIs, can be intercepted by preloading shared libraries. The mechanisms and performance of network bandwidth control are identical across the Windows NT and Linux platforms, so we only discuss how CPU and memory resources are constrained. These resources present challenges different than on NT. The implementation and measurements below refer to Linux kernel version 2.2.12.



(a) (b)

Figure 10: (a) Control of CPU shares on Linux. (b) Application execution time as CPU share varies.

6.1 Constraining use of CPU resources

Monitoring progress Similar to the NT implementation, the CPU monitor is periodically triggered using a 10 ms timer interval. Information about process state (ready or waiting) and accumulated CPU usage are readily available from the `/proc` file system.

Controlling progress Although adjusting the scheduling priority of an application does constrain its CPU usage, this cannot be used on Linux due to the limitation that an ordinary user can only lower an application's priority. Only a superuser can both lower and raise priorities. Instead, we use a different strategy to control application progress, relying on POSIX signals to stop and resume process execution. Upon detecting that an application has received more than its share, the monitor process stops its execution by issuing a SIGSTOP signal. SIGSTOP cannot be caught or ignored; it always stops the target process. When the application's CPU usage dips below the threshold, application execution is resumed using a SIGCONT signal. This control can be performed at fine granularity and incurs low overheads when the time slots are on the order of a millisecond. Note that virtually identical control is possible using the `ptrace` mechanism.

Effectiveness of the sandbox Figure 10 shows the application behavior when CPU share is subject to various constraints. In Figure 10(a), three applications are started under sandboxes,which restrict their CPU shares to be 50%, 30%, and 10%, respectively. The actual CPU resource allocation is obtained by polling the `/proc` file system. After an initial perturbation (due to the introduction of new polling processes), all three applications receive a stable CPU share as requested, contributing to a total system load of over 90%. Figure 10(b) shows the impact of the sandboxing code on execution time as CPU share constraints vary from 5% to 100%. We normalize execution time assuming perfect slowdown in proportion to the requested share, and compare this with the measured execution time. As the figure shows, the two are virtually indistinguishable with the largest error being less than 1%, indicating the steady control possible under the Linux sandbox implementation.

6.2 Constraining use of memory resources

Monitoring progress Similar to the CPU case, the `/proc` filesystem provides the monitoring code with information about the process resident size.

Controlling progress Linux provides a `setrlimit` API for limiting the maximum amount of physical

memory a process can use. However, current kernels do not enforce this constraint. Consequently, we need to adopt a scheme identical to the one on NT, where control of memory usage is accomplished by having the application relinquish surplus memory pages. Protection fault handlers are used to sample the progress metric as well as determine when to evict a page to reduce the working set size. The primary difference stems from the fact that unlike NT, where an implicit protocol (using `NoAccess` protection bits) between the OS and the application permits the former to collect pages not required by the latter, no such protocol exists on Linux. The protection bits can be set as on Linux, but the kernel swapper (`kswapd`), does not check the page attributes to decide which page must be swapped out.

We get around this problem by effectively handling the swapping ourselves. First, we intercept memory allocation functions (e.g., malloc) to make sure that only the requested amount of physical memory is kept valid; all other memory pages are protected to be unavailable for access. When a page fault happens due to invalid access, we pick another page (in FIFO order) from the resident set (maintained by the sandboxing code), save its contents to our own swap file, and taking it out of the resident set using the `munmap` mechanism in Linux. Subsequently, an invalid access requires that the saved contents be mapped back to the corresponding virtual address.

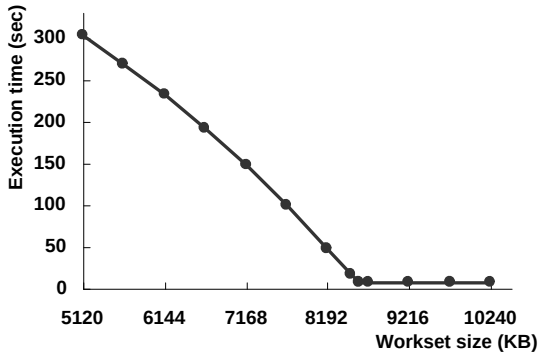


Figure 11: Control of memory usage on Linux.

Effectiveness of the sandbox Our preliminary results show that we can control memory usage to within 3% of the requested amount. Figure 11 shows the execution time of an application (same as the one described earlier for NT) as the amount of physical memory varies. When memory constraint is above its actual usage (8588 KB), no page fault is triggered; the execution time is the same as sandbox-free execution. When memory is constrained to be smaller than what is required, execution time increases due to page faults, but linearly as expected for the application. As with NT, the slope of the execution time curve can differ in different regions of available physical memory.

7 Extensibility Advantages of User-level Sandboxing

This paper has described a general, secure, user-level strategy for building sandbox environments that can impose both qualitative and quantitative restrictions on resource usage by application components running within the sandbox. The description in Sections 3-6 implicitly assumed that the granularity at which the restrictions were being enforced was a single process, or a collection of processes. To counter the argument similar sandboxing mechanisms can be constructed within the kernel instead of at the user level, two points need to be stressed.

First, it is interesting to observe that modern OSes provide sufficient support to permit implementation of both qualitative and quantitative constraints at the user level. The shared library support enables interception of system APIs; the monitoring infrastructure makes possible acquiring of almost all necessary information; the priority-based scheduling, debugger processes, and signal handling mechanisms allows the adjustment of an application's CPU usage; the memory protection and memory-mapped file mechanisms permits control of the amount of physical memory available to an application. Finally, the socket interface gives direct control of network activities. Most resources in an operating system can benefit from some combination of these techniques.

Second, and perhaps more important is the fact that user-level approaches provide more flexibility in deciding the granularity, the policies, and monitoring/controlling mechanisms available for enforcing sandbox constraints. Specifically, it can permit control over resource usage at granularities smaller than what is possible using kernel-level mechanisms.

We demonstrate this extensibility by customizing our process-level sandbox implementation on Windows NT to limiting resource usage at the level of thread and socket groups. The required modifications were trivial, just involving changes in the progress expressions used in the monitoring code and some specialization of the controlling code.

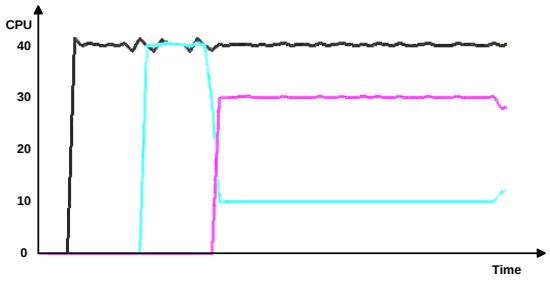
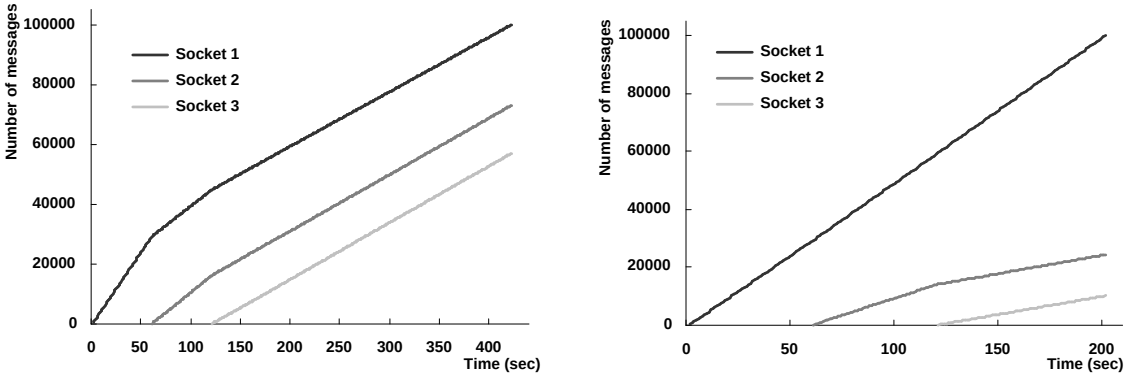


Figure 12: Control of CPU usage at the level of thread groups.

Controlling CPU usage at the thread group level Figure 12 shows a snapshot of the system CPU usage (as measured by the NT Performance Monitor) for an application with two groups of threads, each of which is constrained to a total CPU share of 40%. The application itself consists of three threads which start at different times. Initially, the first thread starts as a member of thread group one and takes up a 40% CPU share. The second thread starts after ten seconds and joins thread group two. It also gets 40% of the CPU share, the total capacity of this thread group. After another ten seconds, the third thread joins thread group two. The allocation for the second thread group adjusts: the third thread gets a 30% CPU share and the second thread receives a 10% CPU share, maintaining the total CPU share of this thread group to 40%. Note that the CPU share of the first thread group is not affected, and that we are able to control CPU usage of thread groups as accurately as at the process level. Currently, the resource allocation to threads in the same group is arbitrary and not controlled. However, one could set up a general hierarchical resource sharing structure, attesting to the extensibility advantages of the user-level solution.

Controlling network bandwidth at the socket group level Figure 13 shows the effect of restricting network bandwidth at the level of socket groups. In this case, the total bandwidth of a socket group is limited to a particular value. The application used in the experiment consists of one server instance and three client instances. The server spawns a new thread for each client, using a new socket (connection). The communication pattern is a simple ping-pong pattern between the clients and the server. Figure 13(a) shows the performance of server threads when the bandwidth constraint is enforced at the process level. The



(a) (b)

Figure 13: Control of network bandwidth (a) at process level, (b) at socket group level.

total network bandwidth is restricted to 6 MBps. The clients and server exchange 100000 4 KB messages. The figure shows the number of messages sent by the server to each client as time varies. The first client starts about the same time as the server and gets the total bandwidth of 6 MBps (as indicated by the slope). The second client starts after one minute, sharing the same network constraint. Therefore, the bandwidth becomes 3 MBps each. The communication is kept at this rate for another minute until the third client joins. This makes all three of them transmit at a lower rate (2 MBps). As a result, the first client takes more than 400 seconds to complete its transmission, due to the interference from the other two clients.

Figure 13(b) shows the case where the first client needs to receive a better and guaranteed level of service. Two socket groups are used, with the network bandwidth of the first constrained to 4 MBps and that of the second group to 2 MBps. Clients start at the same times as before. However, the performance of the first client is not influenced by the arrival of the other two clients. Only the two later clients share the same bandwidth constraint. As a result, the first client takes only 200 seconds to finish its interactions.

These experiments demonstrate that user-level sandboxing techniques can be used to create flexible, application-specific predictable execution environments for application components of various granularity. As a large-scale application of such mechanisms, we have exploited these advantages in other work [5] to create a cluster-based testbed that can be used to model execution behavior of distributed applications under various scenarios of dynamic and heterogeneous resource availability.

8 Conclusion

This paper describes the construction of a secure, user-level resource-constrained sandbox, which exploits widely available OS features to impose qualitative and quantitative restrictions on an application's resource usage. It demonstrates two concrete implementations on Windows NT and Linux operating systems, using three representative resource types (CPU, memory, and network) as example. Our evaluation shows that the user-level sandboxing approach can achieve accurate quantitative restrictions on resource usage with minimal run-time overhead, and can be easily extended to support application-specific constraining policies.

Acknowledgments

We are indebted to Zvi Kedem, who suggested using different priority levels to efficiently control a sandboxed process' CPU usage. We are grateful to Anatoly Akkerman and Arash Baratloo for their help on implementing the sandbox on Linux. In addition, we would like to thank Lionell Griffith for giving us his implementation of fine-grained timers on Windows NT. This research was sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement numbers F30602-96-1-0320 and F30602-99-1-0517; by the National Science Foundation under CAREER award number CCR-9876128; and Microsoft. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

References

[1] K. Arnold and J. Gosling. *The Java Programming Language (2nd Edition).* Addison-Wesley, 1998.

[2] R. Balzer and N. Goldman. Mediating connectors. In *ICDCS Workshop on Electronic Commerce and Web-based Applications*, 1999.

[3] B. Bershad, S. Savage, P. Pardyak, E. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *15th ACM Symposium on Operating System Principles*, 1995.

[4] F. Chang, A. Itzkovitz, and V. Karamcheti. Securing user-level sandboxes. In Preparation.

[5] F. Chang and V. Karamcheti. Automatic configuration and run-time adaptation of distributed applications. Technical Report TR1999-793, New York University, 1999.

[6] A. Chien and J. Kim. Approaches to quality of service in high performance networks. In *Proc. of Parallel Computer Routing and Communication Workshop*, 1997.

[7] A. Demers, S. Keshav, and S. Shenkar. Analysis and simulation of a fair queueing algorithm. In *Proc. SIGCOMM '89 Symposium*, Sep. 1989.

[8] D. Evans and A. Twyman. Flexible policy-directed code safety. In *IEEE Symposium on Security and Privacy*, 1999.

[9] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications. In *Proc. of 6th USENIX Security Symposium*, Jul. 1996.

[10] P. Goyal, X. Guo, and H. Vin. A hierarchical CPU scheuler for multimedia operating systems. In *Proc. of 2nd Symposium on Operating Systems Design and Implementation*, 1996.

[11] L. Griffith. Precision NT event timing. Windows Developer's Journal, Jul. 1998.

[12] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *Proc. of 3rd USENIX Windows NT Symposium*, Jul. 1999.

[13] M. Jones, P. Leach, R. Draves, and J. Barrera. Modular real-time resource management in the Rialto operating system. In *Proc. of 5th Workshop on Hot Topics in Operating Systems*, May 1995.

[14] T. Lindholm and F. Yellin. *The JAVA virtual machine specification.* Addison Wesley, 1996.

[15] C. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proc. of IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[16] George Necula. Proof-carry code. In *Proc. 24th ACM Symposium Principles of Programming Languages*, 1997.

[17] VeriSign. www.verisign.net.

[18] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *14th ACM Symposium on Operating Systems Principles*, 1993.

[19] C. Waldspurger and W. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proc. of 1st Symposium on Operating Systems Design and Implementation*, Nov. 1994.

[20] L. Zhang. Virtual clock: A new traffic control algorithm for packet switched networks. In *Proc. ACM Trans. on Computer Systems*, May 1991.