User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement[*]

K. Jain
Iowa State University
Ames, IA 50014
`kapil@cs.iastate.edu`

R. Sekar
State University of New York
Stony Brook, NY 11794
`sekar@cs.sunysb.edu`

Abstract

Several new approaches for detecting malicious attacks on computer systems and/or confining untrusted or malicious applications have emerged over the past several years. These techniques often rely on the fact that when a system is attacked from a remote location over a network, damage can ultimately be inflicted only via system calls made by processes running on the target system. This factor has lead to a surge of interest in developing infrastructures that enable secure interception and modification of system calls made by processes running on the target system. Most known approaches for solving this problem have relied on an in-kernel approach, where the interception mechanisms as well as the intrusion detection/confinement systems are implemented within the operating system kernel. We explore an alternative approach that uses mechanisms provided by most variants of the UNIX operating system to implement system call interposition at user level, where the system calls made by one process are monitored by another process. Some of the key problems that need to solved in developing such an approach are: providing adequate set of capabilities in the infrastructure, portability of the security enhancements and the infrastructure itself across different operating systems, and minimizing performance overheads associated with interception for a wide range of applications. We present a solution that satisfactorily addresses these issues, and can thus lead to a platform for rapid development and deployment of robust intrusion detectors, confinement systems and other application-specific security enhancements.

1 Introduction

One of the biggest problems faced by businesses, individuals and organizations is the protection of their networked computing systems from damage due to malicious attacks launched remotely via the Internet. Such attacks may utilize a variety of techniques, e.g., exploit software errors in privileged programs to gain `root` privilege, exploit vulnerabilities in system configuration to access confidential data, or rely on a legitimate system user to download and run a legitimate-looking Trojan Horse program that inflicts damage. *Intrusion detection* refers to a broad range of techniques that have been developed over the past several years to protect against malicious attacks. A majority of these techniques take the passive approach of offline monitoring of system (or user) activities to identify those activities that deviate from the norm [1, 26, 10, 12, 17, 21] or are otherwise indicative of attacks [14, 18, 22]. More recently, several proactive approaches have emerged. These approaches can *prevent* or isolate attacks before any damage is caused [9, 13, 20, 28].

Most approaches aimed at preventing intrusions [9, 13, 20, 28] are based on the following observation about attacks: regardless of the nature of an attack, damage can ultimately be effected only via system calls made by processes running on the target system. It is thus possible to identify (and prevent) damage if we can monitor every system call made by every process, and launch actions to preempt any damage, e.g., abort the system call, change its operands (e.g., open a different file from one that is specified) or even terminate the process.

In addition to the preventive approaches, system call interception can significantly enhance the power and effectiveness of most offline intrusion detection techniques that make use of system audit data. This is because system audit logs often do not provide all of the information needed for intrusion detection. For instance, we may need to know the value of a string that was read from a network connection, the target of a symbolic link or contents of a file. While it may be possible to identify and record all such potentially useful information in the audit log, such an approach is likely to be impractical due to excessive overheads involved. Consequently, even offline techniques can benefit from active system call interception, as it enables them to access all the data needed for identifying intrusions, without incurring the overhead for accessing irrelevant information.

For reasons mentioned above, the problem of develop-

[*]This research is supported in part by Defense Advanced Research Agency's Information Technology Office (DARPA-ITO) under the Information System Survivability program, under contract number F30602-97-C-0244.

ing infrastructures for active interception and modification of system calls has attracted a lot of research attention [9, 11, 13, 20]. Modification is effected by user-specified code fragments (sometimes referred to as *wrappers* or *extensions*) that are interposed at the system call entry and exit points. These research efforts have demonstrated that such extensions can be used to enhance application functionality in a variety of ways, e.g., application-specific access control, intrusion detection, transparent enhancements for security (e.g., data encryption) or fault-tolerance (e.g., data replication). It is important to note that all of these enhancements can be obtained using system call interposition *without making any changes to the applications* themselves.

This paper presents a new approach for implementation of a system call extension infrastructure. Our approach is characterized by a user-level implementation like [13], where the system calls performed by one process are intercepted (and possibly modified) by another process. This approach contrasts with a majority of approaches in this area that employ an in-kernel implementation of system-call interposition. Our work improves upon [13] by (a) providing a more extensive set of capabilities for extension code, (b) developing an architecture and implementation that is easily ported to different versions of the UNIX operating system, and (c) presenting a comprehensive evaluation of the performance overheads associated with extensions. Below, we describe previous work in system call interposition and summarize the main benefits of our approach.

1.1 Previous Work in System-Call Interposition

In this section we review prior research in system call interposition, motivate user-level implementations of system call interception, and summarize the research problems that have not yet been addressed in this context. A more extended treatment of related work can be found in Section 8.

Some of the earlier research efforts in system call interposition such as [16, 15] were implemented within libraries. For instance, system calls are accessed via a wrapper function within the `libc` library on most UNIX systems. By linking to a different library that contained modified versions of these wrapper functions, one could effect system call interposition. This approach has the benefit that it is easy to implement and very efficient. An important drawback of this approach is that these wrapper functions can be bypassed; for instance, it is possible for a program to directly invoke the system call using a lower level mechanism such as execution of a software interrupt. Thus this approach is not suitable for security-related applications such as intrusion detection and confinement.

An alternative approach that does not suffer from the above drawback is a kernel-based implementation. In particular, the system call interception is implemented within the operating system kernel, and all of the extension code (for all processes being monitored) runs in the kernel mode. This approach has been adopted by most researchers in this area [9, 11, 20, 28]. One of the primary advantages of a kernel-based approach is low interception overhead. The overheads for system call interposition are determined almost entirely by the extension code. Moreover, a kernel-based implementation offers more power in terms of what can be done within extension code, e.g., extensions can be executed with same process context as the process being monitored. However, the power afforded by kernel-mode operation brings some serious drawbacks as well:

- normal protection mechanisms that guard against errors in one process from damaging another process do not apply for code executed in kernel-mode. Hence it is possible for errors in the extension code for one process to corrupt the memory or files used by another process, or even worse, bring down the entire system. Unless we are extremely careful, there is thus a potential for making the system *less secure* by adding extensions!

- state-of-the-art in kernel-resident software development lags user-level software development significantly. This makes it much more cumbersome and error-prone to write the extension code.

- addition of code at the kernel-level will require superuser privileges. Thus, it is difficult in a kernel-based approach for normal users to develop or deploy their own extensions.

- kernel-based implementations require kernel modifications, which may be viewed as too risky and hence not widely accepted.

These factors motivated [13] to develop a user-level implementation of an infrastructure for confining the actions of helper applications launched from a web-browser or a mail reader. Specifically, they used the capabilities provided by the Solaris operating system to intercept the system calls made by a helper application (i.e., a monitored process) within a monitoring process. Typically, the monitoring process runs with the same privileges as the monitored process. However, due to their focus on helper applications on the Solaris operating system, several important problems such as the portability, expressive power and performance of user-level system call interposition were not explored. For instance, they study performance of CPU-intensive code that makes few system calls, e.g., `ghostscript` and `mpeg_play`. Overheads for system-call intensive applications such as network and file servers was not studied. In addition, their focus on application-specific access control requires only limited capabilities for the extension code,

Application	Real time (Low load)	Real time (High load)
gzip	<2%	<2%
ghostscript	<5%	<10%
tar	5%	10%
cp -r	5%	10%
ftpd	<2%	30%
httpd	<5%	35%

Figure 1. Overhead for system call interception for different applications.

e.g., no attempt was made to modify system call data or return code. Finally, they did not address portability of their system to other popular variants of UNIX such as Linux. Consequently, a comprehensive treatment of the issues in user-level implementation of system call extension infrastructures has remained open. We address this problem in this paper and show that we can in fact build powerful and efficient infrastructures at the user level. The key contributions of this paper are summarized below.

1.2 Summary of Results

In this paper, we present an approach for implementing a system-call interposition infrastructure at the user level. The key issues addressed in this paper are:

- *Portability of extension code:* Clearly, an extension cannot be used across different UNIX variants if it makes use of features unique to one of these variants. One approach to make extensions portable is to restrict the interface provided by the interposition infrastructure so that it only exposes features common to all UNIX variants. However, such an approach is unnecessarily restrictive. A better alternative is to permit extensions to access system-specific features, yet assure that those extensions that operate using only those features common across UNIX variants can be used "as is" on these variants.

  We tackle this problem by designing an object-oriented interface that encapsulates OS-dependencies (in terms of system call names as well arguments) so that they are hidden from extension code. Moreover, our approach enables similar system calls (e.g., the many different system calls for reading from a file) to be grouped together so that they can be handled in the same way. By varying the grouping as needed for different UNIX variants, it is possible to write extension code that can be used without any modifications across these variants. A detailed description of our approach in this context can be found in Section 3.

- *Portability of the interception infrastructure:* Most modern versions of the UNIX operating system such as Solaris, Linux, IRIX and OSF/1 provide a mechanism for one user process to trace another process with the same userid, or if the first process has the permissions of the superuser. An user-level infrastructure can be implemented using this mechanism. However, since this mechanism is intended primarily for debugging, it does not operate in the same way on different UNIX variants, thus posing a challenge in terms of portability of the infrastructure. We tackle the portability problem by partitioning the infrastructure into an OS/architecture-dependent component and a second component that is independent of them. The OS-independent component comprises the bulk of the functionality of our infrastructure. It uses the primitive capabilities provided by the OS/architecture-dependent component that may have to be implemented differently on different OS's and processor architectures. Our approach for making infrastructure portability is detailed further in Section 4.

- *Capabilities of extension code:* even though most operating systems provide an ability to intercept system calls, they do not always provide adequate or convenient mechanisms for accessing or modifying the system call data. For instance:

  - Some UNIX variants such as Linux do not provide a way to abort system calls, yet we need these capabilities to confine applications or to prevent intrusions.
  - Modification of variable-size data such as strings poses a problem, as the new value may occupy more space than what was originally available. Thus, the obvious approach of overwriting the data will not work.
  - Malicious applications may attempt to modify system call arguments between the time they are checked by the monitoring program and the time the system call is executed. (Such changes are possible if the malicious program is multi-threaded.)
  - Since the monitoring process and monitored process typically run with the same privileges, we need to guard against a monitored process from interfering with the operation of a monitoring process, e.g., by attempting to kill the monitoring process or reduce its priority.

  Our approach for addressing these problem is also described in Section 4.

- *Efficiency:* As compared to a kernel implementation, a user-level implementation of system call interception

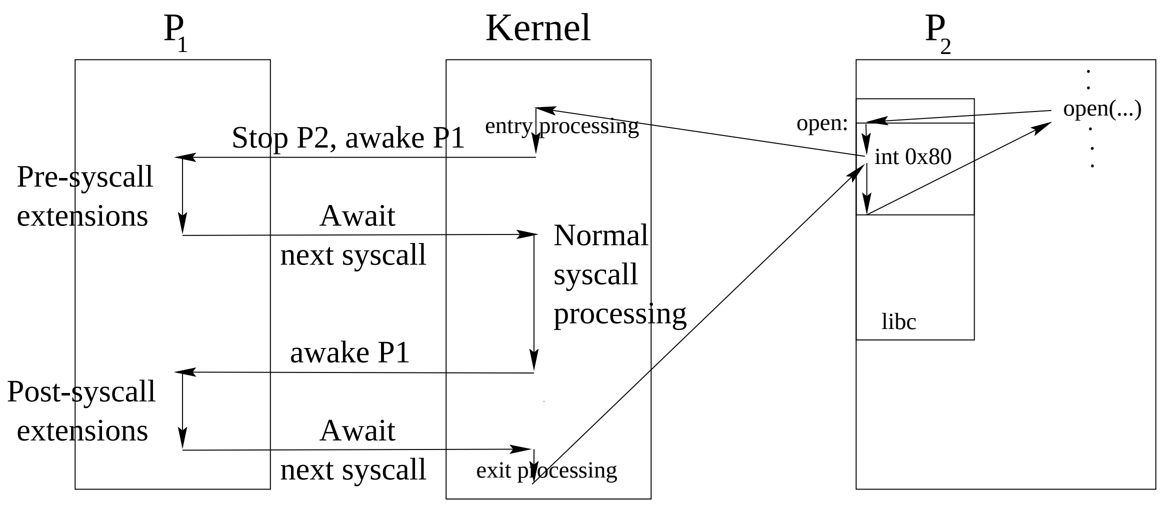

Figure 2. System call execution sequence when a process is monitored.

incurs the following additional costs. First, two additional context switches are introduced for each system call in a user-level implementation. Second, the operations to access or modify the memory of the monitored process are generally less efficient than those available to a kernel implementation. To obtain acceptable performance, we adopt the following techniques. First, we employ the fastest mechanism available under each OS for implementing the memory access operations. Second, we make use of selective interception capability (where a selected subset of the system calls are intercepted) to minimize context switches on OS's that provide this capability. Finally, we use lazy dereferencing of accesses to the monitored process memory, i.e., operations to access system call arguments are deferred until their values are really needed. Using these techniques, we have been able to obtain good performance for user-level system call interception, as described further below.

- *Performance analysis:* For applications such as intrusion detection and confinement, the overhead due to context switch (typically in the range of the time needed to execute thousands of instructions) far outweighs the overhead of memory access operations or executing the extension code [25]. (See Section 5 for details.) Thus our performance analysis in Section 6 focuses primarily on the overhead due to interception alone, while leaving out the overheads due to execution of extension code. (Measuring the overhead due to extension code execution is not useful for a comparison of in-kernel versus user-level implementation of system call interposition.) Figure 1 summarizes our performance results on a 350MHz Pentium II PC running RedHat 5.2 Linux. The table shows the overhead in terms of increase in perceived execution time (also known as real time). For measuring real time, light load corresponded to one instance of a process in the case of `gzip`, `ghostscript`, `tar` and `cp -r`, and a throughput of 20Mb/s for `httpd` and `ftpd`. A high load corresponds to ten instances of the processes (for `gzip`, `ghostscript`, `tar`, `cp -r`) and over 100Mb/s throughput for the servers. Our results show that for CPU and disk-intensive applications, the overheads due to system call interception are very small. Even for servers, where the overhead is moderate under high loads, we believe that the overheads are well-worth the increase in security that can be obtained using system call interposition.

We note that our approach for tackling the first two issues are largely applicable to kernel-based implementations as well.

2 System Overview

Almost all versions of UNIX provide a mechanism for one process to trace and control the execution of another process and/or access its memory using the system call `ptrace`. The primary use of this mechanism has been in implementing debuggers. Some UNIX variants such as Linux provide an enhancement to this mechanism that enable one process to trace the system calls made by another. System V Release 4 (SVR4) compatible versions of UNIX support a more powerful and convenient mechanism for system call interception via the `/proc` interface.

The sequence of actions involved in system call tracing is shown in Figure 2. A user process P_2 invokes a sys-

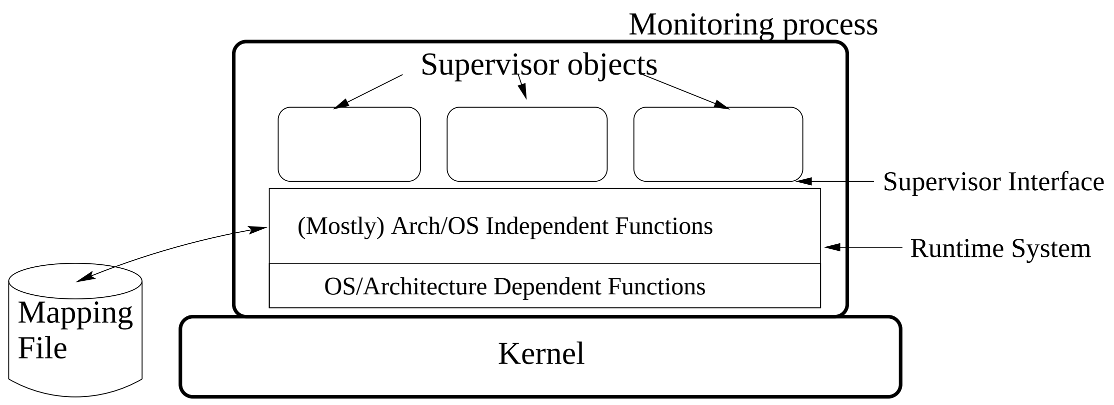

Figure 3. Components of the system

tem call such as `open`, which refers to an entry point in `libc`. After setting up the arguments and the system call number in appropriate registers, this code transfers control into the kernel by using a processor trap or software interrupt (e.g., `int 0x80` in Linux/i386). In the kernel mode, the entry processing code wakes up the monitoring process P_1, which can now examine or modify the registers or the memory of P_2. After whatever processing desired by P_1, it transfers control back to kernel, where the actual system call functionality is performed. Before exiting the system call, control is transferred back into P_1, where additional actions such as examination/modification of the system call return parameters and return code may be performed. Finally, control is returned back to P_2.

In order for P_1 to trace P_2, the effective userid of P_1 must be `root`, or the same as the effective userid of P_2. One of the primary benefits of a user-level interception approach is that the monitoring process does not need the privileges of superuser. This means that ordinary users can develop and use system call extensions in ways they choose. Equally important, the potential damage caused by the monitoring process can be no more than what can be caused by the monitored process. Note that this is not the case with a kernel-based implementation of interposition, where (faulty) extension code can cause damage that is beyond the damage that can be caused even by a root-owned process.

Our system call interception/extension infrastructure is built on top of the system call tracing facilities provided by the different UNIX variants. The overall architecture of our system is shown in Figure 3. An instance of the infrastructure resides within a single user process that we call *monitoring process*. A single process may monitor (and control the behavior of) one or more processes. The actual code for monitoring a process resides in a *supervisor object*, which in our implementation is a C++ object. There will be several monitoring processes running on a system, each monitoring one or more processes. (There will likely be at least as many monitoring processes as the number of distinct users that have processes running on the system.)

All supervisor classes are derived from the base class `SupIfc`. Communication between the runtime system and supervisor object happens via method invocations on this interface. The supervisor classes are dynamically loaded into the monitoring process as new processes are identified for monitoring. A mapping file maps a fully-qualified executable name into a shared library containing the code for a supervisor class. This provides a (static) mechanism to identify which supervisor class will be used to monitor a process. A second (and more dynamic) mechanism provides for a parent process to determine which supervisor object will be used for monitoring its child.

The supervisor objects reside on top of a runtime support infrastructure which consists of two parts. Lower part of the runtime system consists of code that is largely specific to different operating systems and architectures. The differences here arise not only due to the differences in system call interception mechanisms (i.e., variations of `ptrace` or `/proc` interface), but also because these interfaces do not typically provide all of the capabilities needed for modifying the behavior of system calls. As such, our implementation has to access registers and other architecture-specific aspects of the system. The rest of the runtime system consists of code that makes use of the OS/architecture-specific functions. This code remains the same across different architectures and operating systems, except for minor variations due to the fact that the system call names and arguments vary slightly across different UNIX variants.

3 Supervisor Interface

The supervisor interface is designed so that supervisor classes can be written without having intimate knowledge of

1. how system call interception is performed
2. internal codes used to identify different system calls
3. architecture/OS-specific ways to access system call arguments or return value
4. OS-specific mechanisms to read or modify monitored process data

In addition, the interface should provide:

5. support functions to control execution of the monitored process
6. abstraction mechanisms for writing a supervisor class without having to hard-code system call names or argument types specific to one or more variants of UNIX

Our first step is to define a base class called `SupIfc` from which all supervisor classes are derived. For each system call there are two methods in the interface corresponding to the entry and exit for the system call. For instance, corresponding to the `read` system call, the following methods are defined in `SupIfc`:

```
void read_entry(Integer& fd, CharPtr& buf,
                Integer& count)
void read_exit(Integer& fd, CharPtr& buf,
               Integer& count, Integer& rv)
```

Whenever a monitored process P enters or exits a system call (say, `read`) the runtime system identifies the supervisor object S_P monitoring P and invokes the corresponding entry or exit (i.e., `read_entry` or `read_exit`) method on it. This approach addresses issues (1) and (2) mentioned above. To address issue (3), the system call arguments and return value are passed explicitly as arguments to the entry or exit methods.

Issue (4) is addressed by encapsulating the value of each system call argument in an object. This frees the supervisor object from having to deal with the details of layouts of system call arguments data on a particular OS/architecture, or the mechanisms to be used to read the memory of the monitored process. Instead, it relies on methods provided by the objects to conveniently examine or modify the fields of system call arguments. Finally, encapsulation of system call arguments in class objects enables lazy dereferencing of arguments. For instance, an `Integer` object can be constructed with a field recording the register in which the argument is located. If the value of the register is needed, the supervisor method will invoke a `get` operation on it, at which point the register contents can be fetched. As shown in Section 6, this lazy dereferencing will enhance system performance on some operating systems.

To illustrate system call argument classes, we give an example of the `CharPtr` class below:

```
class CharPtr {
  int get(char *buf, int len);
  int put(char *buf, int len);
  int lockVal(); /* ensure that value can't
    be modified by a thread of the monitored
    process before syscall completion */
}
```

The `get` and `put` have obvious meanings. To understand the `lockVal` operation, note that if the system call argument is from a region of memory shared by multiple threads or processes, it may be possible for one of the threads/processes (other than the one making the system call) to modify the contents referenced by a pointer argument between the time it is examined by the supervisor and the time it is fetched by the kernel when the system call is executed. The `lockVal` method is used to indicate to the runtime system that this must not happen. The implementations of these system call argument classes is provided by the runtime system, as described in the next section.

Issue (5) is addressed by defining a symmetric interface to `SupIfc` called `RtIfc` that defines several support functions such as:

- `kill` sends the specified signal to the monitored process.
- `abort`, which can be invoked only from within a method invoked on system call entry, prevents the system call from completing. An argument to `abort` will specify the value to which the return code should be set to before returning control to the calling program.
- `switch` action replaces the current supervisor object with a new object belonging to the class specified as the argument. This ability to dynamically change the supervisor object is particularly useful when we suspect (or know for a fact) that a monitored process is misbehaving, and we wish to monitor the process more closely (or even better, confine its actions). Moreover, the switch capability is useful to implement sophisticated policies regarding which supervisor object should monitor a process after it `execve`'s another program.

The approach presented so far has not addressed issue (6), since it requires the supervisor class to support methods with names that exactly match the system call names in a particular (version of an) operating system. To overcome this problem, we have developed a higher level language that provides an abstraction by which groups of related system calls can be abstracted into a higher level event. This abstraction mechanism is one of the features of our behavioral specification language for intrusion detection/prevention described in [26]. The basic form of an abstract event definition is:

$$eventName(\overline{X}) ::= (s_1(\overline{Y}_1)|Cond_1)|| \cdots ||(s_n(\overline{Y}_n)|Cond_n)$$

We have used the shorthand $\overline{X}$ to denote a list $x_1, ..., x_k$ of arguments, and $s_1, ..., s_n$ denote either a system call or another abstract event. (But cyclic definitions are not permitted.) Each $Cond_i$ provides a binding for each of the variables in $\overline{X}$ in terms of the values of one or more variables in $\overline{Y}_i$. Additionally, $Cond_i$ may consist of conditions on the variables in $\overline{Y}$. The condition and expression syntax are similar to that of C++.

An abstract event that denotes operations to read a file using a file descriptor operand can be defined as follows for Linux:

```
readFd(Integer fd) ::=
  read(fd,_,_)||readdir(fd,_,_)||
    getdents(fd,_,_)||readv(fd, _, _)
```

Here, "_" denotes an argument whose value is not of interest for defining the abstract event. The same abstract event may be defined differently for OSF/1:

```
readFd(Integer fd) ::=
  read(fd,_,_)||pread(fd,_,_)||
    getdirentries(fd,_,_,_)||readv(fd,_,_)
```

Predefined collections of such abstract events can be specified in *interface definitions*, which serve a purpose similar to OS-specific header files. The supervisor class can now be defined in an operating system independent fashion by providing a method for each of the abstract events in the interface definition, e.g., `readFd`.

A compiler for the interface definitions is responsible for arranging to call the code corresponding to the abstract event $eventName$ whenever any of $s_1, ..., s_n$ are called, and moreover, the corresponding condition is satisfied. The code for $eventName$ is invoked with parameter bindings as given by the condition. In effect, the compiler generates a stub routine for each system call that in turn ends up calling an operation in the supervisor class with the name corresponding to the abstract event[1].

We note that [9] uses a related approach for making their wrappers portable. Our interface definition files are similar to their *characterization* files. They make use of a notion of *tagging* system calls, and use the tags in a way similar to our abstract event names. An advantage of our approach is that event abstractions can be composed, while tagging is not.

[1]If multiple abstract events match the same system call, all of them are invoked in some order. Undesirable interactions may arise due to interference among actions for these different abstract events – this is an area of current research.

3.1 Example Supervisor Class

We provide an example of a supervisor class `UntrustedUtility` to illustrate the API described so far. We describe a simple class which restricts the monitored process from performing many damaging system calls, and also restricts the files that can be opened by the application. We begin by defining abstract events of interest:

```
deniedCalls ::= fork||execve||connect||bind
  ||listen||chmod||chown||chgrp||kill||ptrace
  ||sendto||mkdir||utimes||rename||...
  /* other disallowed calls omitted */
wrOpen(f) ::= (open(f,md)|isWrite(md))
  ||creat(f)||truncate(f)
```

We have omitted some or all of the trailing arguments of some system calls in the above definition for conciseness. After defining the abstract events, we can then provide the implementation of just two methods, `deniedCalls` and `wrOpen`. Any system call that does not match one of these abstract event definitions will be given the default treatment, which for most system calls is to continue. (The runtime will include certain protection mechanisms that ensure that the monitored process cannot make system calls such as `kill` on the monitoring process.) The implementations of these two methods may be as follows:

```
void UntrustedUtility::deniedCalls_entry() {
  abort(EPERM);
}
void UntrustedUtility::wrOpen(CharPtr f) {
  char *fv = f.get();
  /* we may want to lockVal(), first */
  if (!isInDir(fv, "/tmp") && (exists(fv)))
    abort(EPERM);
}
```

The code for `wrOpen` uses support functions to determine whether the file being opened for write is inside the `/tmp` directory, and if not, whether it will end up creating a new file or modifying an existing one.

4 Runtime system

A monitoring process is started with the name of an executable to be monitored. Optionally, the name of the supervisor class to be used for monitoring the executable and the name of the file containing the object code for this class may be specified. Otherwise an appropriate supervisor class is identified from the mapping file. The monitoring process then loads the supervisor class and then forks and uses the `attach` primitive provided by OS-specific module to attach to the child process. Following this, the child process exec's the executable, and now the process is set up to be

```
while there exist processes to be monitored {
   pid = waitForCall(); /* wait for a monitored process to enter/exit sys call*/
   call = getscno(pid);
   if (isEntry(call)) {
      /* Pre-entry processing, details omitted */
      switch (call) {/* get system call identifier */
         case OPEN_ENTRY:
            supObj[id].open_entry(scInfo[OPEN_ENTRY][0],
                  scInfo[OPEN_ENTRY][1], scInfo[OPEN_ENTRY][2]); break;
         /* cases for other system calls not shown */
      }
      /* Post-entry processing (omitted) */
   }
   else if (isExit(call)) {
      /* Pre-exit processing (omitted) */
      switch (call) {/* get system call identifier */
         case OPEN_EXIT:
            supObj[id].open_exit(scInfo[OPEN_EXIT][0], scInfo[OPEN_EXIT][1],
                  scInfo[OPEN_EXIT][2], scInfo[OPEN_EXIT][3]); break;
         /* cases for other system calls not shown */
      }
      /* Post-exit processing (omitted) */
   }
   else if (isSignal(call)) {
      /* signal related processing (omitted) */
   }
}
```

Figure 4. Main loop of the runtime system

monitored by an object of the specified supervisor class. When the runtime system is monitoring one or more processes, it is executing within a loop shown in Figure 4.

The functions `waitForCall` and `getscno` are provided by the architecture/OS-specific component of the runtime system. This component also includes the definition of the array `scInfo` which specifies the registers (or offsets into the kernel-maintained user structure for the monitored process) that contain the system call arguments. System call argument class objects such as `Integer` and `CharPtr` are constructed from these register numbers. (This step happens implicitly, thanks to the type conversion rules of C++.) Finally, the runtime looks up its association table to identify the supervisor object corresponding to the process that made the system call, and the corresponding entry or exit method is invoked on this class. On return from this call, the monitoring process goes back into `waitForCall`.

While the action of the runtime system to most system call entry and exit operations are similar to that for `open`, some operations require special treatment:

- `fork`: the runtime system needs to identify the pid of the child process, and be ready to monitor it. Moreover, the supervisor object itself is cloned, with one copy monitoring the parent process and another monitoring the child.

- `execve`: no special processing is required on the part of the runtime system. If the current supervisor object requires the new process to be monitored by a different supervisor object, it explicitly invokes `switch` operation from its `execve_entry` method with the appropriate shared library name and class name for the new supervisor class. Alternatively, the `switch` operation may specify a mapping file that is looked up to determine the supervisor based on the executable name. Switching to a new object will be done as part of post-supervisor processing.

- `kill`: since the runtime system typically runs with the same userid as the process being monitored, we have to be careful to ensure that the monitored process cannot kill the monitoring process or otherwise damage its ability to monitor it. This is ensured by intercepting every “potentially dangerous” system call such as `kill`, and permitting it to go through only if it will not affect this or other monitoring processes. A similar remark applies to several other system calls that can interfere with the monitoring process.

- `exit`: on completion of the entry function, the supervisor object for the monitored process is destroyed by the runtime system.

4.1 Implementation of System Call Argument Classes

The runtime system also provides the implementation of the system call argument classes. These implementations are made independent of the OS and machine architecture by using the functions provided by the architecture-specific module. The implementation of these classes is mostly routine, except for the way in which we deal with modifications to the memory referenced by pointer arguments, which we discuss further below.

For instance, if we have a filename argument, and would like to modify it, this can be done in place if the new name is smaller than the old one. Otherwise, we need to allocate new storage, and change the system call argument value to point to this new location. The catch is that this new location must be within the address space of the monitored process, and it is difficult to identify unused sections of memory from the monitoring process. Our approach is to allocate new storage on the stack beyond the top of stack. We find that normally, the allowable stack size is much larger (by several MBs) than the value of stack pointer, so all of this space can be used.

Regardless of whether the new value requires more storage space than old value, we may choose to store the new value at a random location on the stack. The motivation for this is as follows. There is a window of time between the assignment of a new value to a system call argument and the time when these arguments are actually fetched from user memory for system call execution by the kernel. In a multithreaded environment, one of the threads can make a system call, while the other tries to locate and modify the system call argument back to its original value. Even worse, the first thread may provide a valid-looking argument (e.g., a file name) which is permitted by the supervisor object, while the second thread changes the file name to an unacceptable value. Putting the modified value at a random location on the stack implies that the rogue thread has to search through the entire stack to identify where the value is stored. If there is sufficient stack space, say 1MB, this may take millions of instructions to complete. Compared with this, the time window being exploited is probably much smaller, say, several hundred instructions. This implies that the probability of the rogue thread succeeding in its effort is very small[2].

[2] A rogue program may try to circumvent this approach by leaving very little room on the stack where the runtime system may store arguments. Since typical programs execute with several MBs of available stack space at all times, we may simply terminate the process when insufficient stack space (say, less than 1MB) is detected.

The race condition with respect to checking of system call parameter values stored within user memory was first addressed in [11]. They identified their ability to deal with the race conditions reliably as one of the benefits of an in-kernel implementation. While a similar guarantee can probably never be provided in a user-level implementation, we believe that approaches such as ours that exploit some sort of randomization can reduce the success probability of such race attacks to a negligible value. Nevertheless, it must be noted that successful attacks can go undetected, in the absence of kernel support.

4.2 Architecture dependent primitives

This module contains the lowest level functions whose implementations will vary significantly across different operating systems and/or processor architectures. The functions provided by this module include:

- `getScNum, isEntry, isExit, isSignal`: get the system call number (or signal received by the monitored process) as an OS-independent value, identify if we intercepted a system call entry, exit or a signal. This module also needs to identify the register numbers where the system call arguments are stored.

- `getReg, setReg`: get or set the values of registers. We note that register numbers do not necessarily correspond to the processor hardware registers. They are simply handles created by the architecture-dependent module to refer to some location within the user structure (or anywhere else within the process control block) that can be read or modified by the architecture dependent module.

- `getData, setData, getText`: Read or modify the memory of the monitored process. Its implementation is different for `ptrace` and SVR4 based OSes. For `ptrace`-based interface, reading or writing bulk data can be inefficient, so we employ any OS or architecture specific enhancements to these capabilities, e.g., we use the `/mem` facility in Linux which permits efficient reads but not writes.

- `attachProcess, waitForCall`: Again, the implementations of these operations are different across `ptrace` and `/proc` interfaces.

- `abort`: SVR4-compatible UNIX versions provide a special operation for aborting system calls. Still, we needed to develop a way to modify the return code so that any value chosen by the supervisor object can be provided to the monitored program, rather than the default value in SVR4 which indicates an error `EINTR`

TestCase	Normal execution time	Time spent in extension	Overhead
ftpd	2.2s	0.03s	1.5%
telnetd	3.1s	0.04s	1.3%
httpd	5.8	0.09s	1.5%

Figure 5. System call interposition overhead.

(system call interrupted). The `ptrace` implementation in Linux provides no way to abort system calls. Our approach is to modify the system call argument to a value that would cause the call to fail. We then intercept the system call exit and set the return values and parameters as appropriate. Identifying such argument values (that cause system calls to fail) is fairly simple in most cases, e.g., null pointer value for file names. However, some calls do not take arguments, so this approach fails. Fortunately, most such system calls either do not change system state (so no harm in letting them complete, plus we can modify the return values), or are not permitted to be executed even by superuser-owned processes (e.g., `setup` and `idle` in Linux). Notable exceptions are `fork` and `exit`. With fork, our approach is to kill the child before it completes its first system call. With exit, it is arguable whether there is any merit in aborting it, so we simply let it complete.

5 An Application in Intrusion Detection and Confinement

We have built a system for intrusion detection, application confinement and application-specific access control based on system call interposition [25, 26]. Our approach is based on specifying security-relevant properties of programs as patterns over sequences of system calls executed by processes. Our specification language enables us to capture conditions on system call names as well as their arguments. Response actions can be associated with each security property, and these responses will be triggered whenever the property is violated. These actions may be used for a variety of purposes such as disallowing the violating system call, modifying its argument so that a different resource from the one specified in the system call is manipulated, or terminate the process. A compiler for this language translates these specifications into C++ code for a supervisor object. This code is compiled with the C++ compiler and then linked with the system call interposition infrastructure to provide intrusion detection and confinement.

The results shown in Figure 5 were obtained with a runtime system that uses in-kernel system call interposition [4]. Thus any overhead measurements relating to system call interception or data access cannot be extrapolated to the user-level interposition approach. However, the results do establish that our approach of achieving portability of system call extensions (using the `SupIfc` interface and the system call argument classes, which are implemented in the same way in both runtime systems) is effective. They also show that the overhead due to the execution of extension code is very small — in fact, it is negligible as compared to the overheads for system call interception and argument access. Consequently, in a user-level approach such as ours, overall performance for applications such as intrusion detection, confinement and access control can be measured purely in terms of the overheads for system call interception and data access. (This is precisely how we analyze performance in the next section.)

The results shown in Figure 5 were taken on 350MHz Pentium II Linux PC with 128MB memory and 8GB EIDE disk. They include only the time spent within the extension code. Sample specification for the `ftpd` server can be found in [25]. This specification consists of approximately 15 properties that restrict the operations of the FTP server so that it (a) accesses only certain files before user login, and certain other files after user login, (b) executes only certain files, (c) performs host and user authentication, (d) allows connections back to the FTP client, but not arbitrary hosts, (e) does not use privileged system calls or system calls except for binding to a privileged TCP port, and (f) enforces additional restrictions for certain users, e.g., anonymous users and super user. The specifications for `telnetd` and `httpd` were similar.

6 Performance Results

The primary goal of our performance experiments is to assess the impact of additional overheads introduced by user-level system call interposition as compared to kernel based interposition. As mentioned earlier, the time spent inside the extension code remains the same for both approaches, but additional overheads are incurred in the user-level approach due to context switches and relatively inefficient operations to access the memory of monitored process. Therefore our experiments were designed with null extension bodies, or extensions that simply access (and possibly modify) system call arguments. For kernel-based implementations, the overheads due to such extensions will be negligible as compared to the overheads for a user-level implementation.

Measuring the overhead due to execution of extension code will serve to measure whether computationally intensive tasks were performed within extension code. Such measurements would be useful for determining the suitability of system call interposition for accomplishing a particular task. Prior research in [9, 20], as well our results [25] summarized in the previous section, provide adequate evi-

Application	CPU time	Real time (Low load)	Real time (High load)
gzip	<2%	<2%	<2%
ghostscript	10%	<5%	<10%
tar	30%	5%	10%
cp -r	50%	5%	10%
ftpd	70%	<2%	30%
httpd	65%	<5%	35%

Figure 6. Overhead for system call interception for different applications.

dence to the efficiency and effectiveness of the system call interposition approach for applications such as intrusion detection, confinement and access control. However, measuring the execution time within extension code will not in any way help assess the performance of the infrastructure developed in this paper. This is another reason for our focus on very simple extensions in our performance study.

Most of the results presented in this section were obtained for Linux running on a 350MHz Pentium II with 128MB memory and 8GB EIDE disk, while the others were obtained on IRIX running on SGI R10000 150MHz 128MB 4GB SCSI, and OSF/1 running on DEC alpha 500MHz with 1GB memory. Whenever the measurements pertain to IRIX or OSF/1, we indicate this explicitly; otherwise the measurements pertain to Linux.

6.1 Overhead for System Call Interception

For performance analysis, we consider three categories of applications to monitor: CPU-intensive, disk I/O intensive, and network-intensive applications. The overhead is measured in terms of increase in CPU time, as given by the formula:

$$\frac{\text{CPU time for monitored app AND monitoring process}}{\text{CPU time for unmonitored app}} - 1$$

CPU time is taken to be the sum of the times spent in user and system mode by the system on behalf of a process. We also measure the perceived degradation in performance, which is computed using real time (otherwise called elapsed time) instead of CPU time. The perceived degradation is shown under lightly loaded and heavily loaded conditions in Table 6. In the table, lightly loaded conditions refer to a single instance of a running application for `gzip`, `ghostscript`, `tar` and `cp -r`, and a delivered throughput less than 20Mb/s for `ftpd` and `httpd`. Heavy load corresponds to running ten instances of an application for `gzip`, `ghostscript`, `tar` and `cp -r`, and a delivered throughput of 100Mb/s or higher for `ftpd` and `httpd`.

The graph in Figure 7 shows the increase in CPU time as a function of the number of system calls made by different applications. We expect the overhead to be mainly

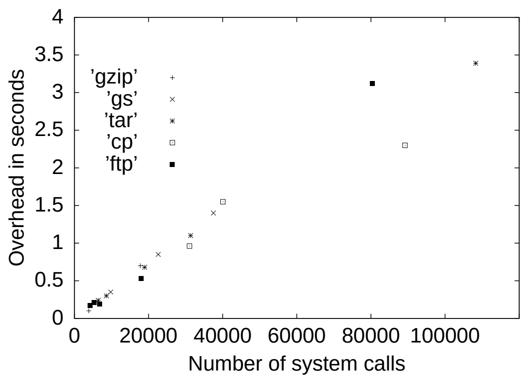

Figure 7. Increase in overhead with increase in number of system calls.

due to context switches, whose cost should be independent of the cost of the system call itself. Thus we expect a linear relationship between the overhead and the number of system calls made. The graph shows that this overhead is between 26 and 38 microseconds per system call, with most points concentrated around 34 microseconds. The variation is a result of time measurement errors which get amplified since we are taking the difference between CPU times with and without monitoring. Moreover, when a process is monitored, we need additional processes for monitoring, which can in turn lead to poorer virtual memory and cache performance. The impact of this is hard to predict, and may vary from application to application.

6.1.1 CPU-intensive applications

This category is meant to represent typical CPU-intensive applications that are used frequently. Users are typically sensitive to additional overheads for such applications, since their running time is long enough to be perceived. We consider `gzip` and `ghostscript` in this category. While `gzip` makes very few system calls, `ghostscript` makes a moderate number due to the socket-related operations needed for displaying on X-windows.

Our overhead results confirm the results of [13] for SVR4 compatible operating systems such as IRIX, while establishing that similar overheads (almost immeasurable) hold for Linux implementation that uses the `ptrace` facility.

6.1.2 Disk I/O-intensive applications

In this category, we studied `tar` and `cp -r` (i.e., recursive copy of directories). This category of applications tends to make a very large number of system calls. Since our main overhead is due to context switching for every system call, higher overheads are to be expected for these programs. However, due to a large amount of I/O operations

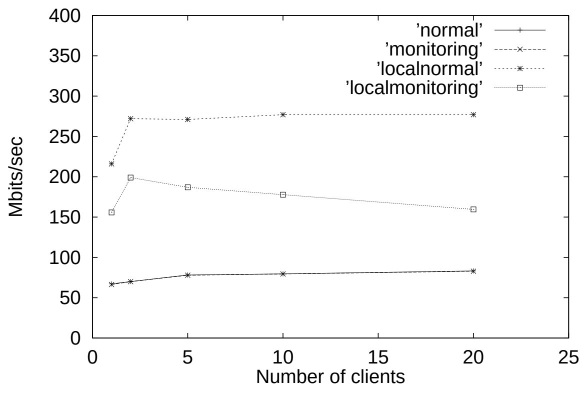

Figure 8. FTP server throughput

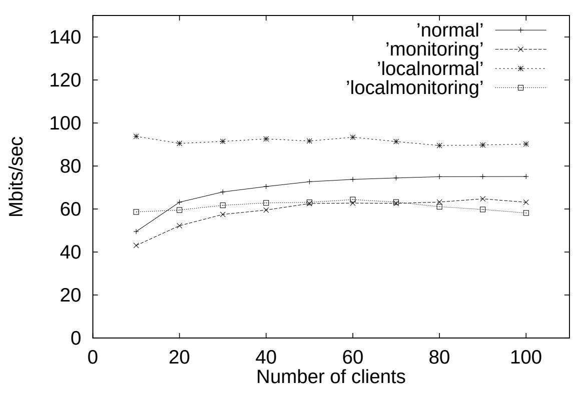

Figure 9. HTTP server throughput

to the disk, the elapsed time is much larger than CPU time, thus the perceived increase in overhead is typically small for these applications. Due to opportunities for multiplexing in I/O, the perceived overhead increases when multiple concurrent instances of these applications are run.

6.1.3 Network Servers

Network servers such as `ftpd` and `httpd` are among the most important applications that need the increased security offered by system call monitoring. Consequently, our performance analysis on these applications is more comprehensive. Rather than measuring just the increase in CPU time due to system call interception, our experiments were geared more towards evaluating the degradation in throughput and latency experienced by clients of these services.

The test suite for `ftpd` consisted of several files whose sizes were uniformly distributed over the range of 0.5 to 5MB. First, we ran the server and client on different machines.We ensured that all of the requested files would together fit within the in-memory file cache for the server, so the effect of disk access was completely eliminated in these tests. Figure 8 shows how the throughput varies as a function of the number of clients. The graph shows that the throughput is limited by network bandwidth, as the curve flattens out around 80Mb/s, which is probably close to the best that can be achieved on a 100Mb/s Ethernet, when all of the protocol overheads are considered. Since the CPU utilization remains fairly low at this point (around 20%), the overheads of monitoring do not affect the throughput at all; thus the throughput with monitoring coincides with the throughput without monitoring.

To simulate what happens under higher CPU loads (or faster networks), we then ran the client and server on the same machine. Moreover, we specified the local file name on the client to be `/dev/null` so as to eliminate the effect of disk access. In this case, the throughput is limited by CPU usage, so there is a significant degradation in throughput.

Figures 9 shows throughput results for `httpd`. The tests were performed using WebStone, which is one of the widely used benchmarks for web server performance. We used WebStone with the standard fileset included in the benchmark. We used two client machines and one server machine. We tested starting with 10 clients going to 100 clients in increments of 10. The time per run was set to 10 minutes and we took 3 iterations. On the server, httpd was run with and without monitoring. WebStone provided us with the results for the throughput and the response times. When the client and server are run on different machines, the throughput saturates at about 75Mb/s when no monitoring is performed. (This number is lower than that for `ftpd`, reflecting a higher overhead for HTTP due to the protocol itself, and due to a different distribution of file sizes.) At the saturation point, the CPU utilization is sufficiently high to result in measurable drop in throughput of about 15%. When the experiments are performed with the client and server on the same machine, the overhead is even more pronounced (about 30 to 35%). WebStone also provides response times for `httpd`, which are shown in Figure 10. (Response times are not very meaningful in our `ftpd` experiments since the large file sizes meant that the time to receive a file was almost completely due to the time for transferring the files.)

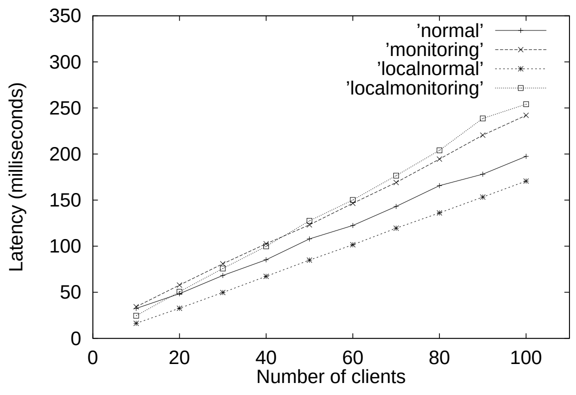

Figure 10. HTTP response time.

Operating System	Intercept only	Access arguments (read-mostly)	Access filenames (read-mostly)	Follow all pointers (read)	Follow all pointers (read/write)
IRIX	55%	55%	75%	330%	400%
Linux	27%	34%	39%	60%	2000%

Figure 11. Overhead for reading and writing system call arguments.

Operating System	Overhead
Linux	37%
IRIX	60%
OSF/1	35%

Figure 12. Interception overhead on IRIX, OSF/1 and Linux.

6.2 Interception Overheads on Other UNIX Variants

Table 12 shows the overheads for system call interception on two variants of System V Release 4 compatible operating systems, namely, Irix and OSF/1. We only show the average of the overheads among the six applications shown in previous tables. These operating systems provide a finer granularity of control over what system calls are to be monitored. We used this capability so that we ignored "uninteresting" system calls like `read` and `write`. In addition, for most system calls, we intercepted only the entry and not the exit. The overheads for interception under these conditions are shown in the table and compared against the times obtained for Linux. Although Linux does not permit selective monitoring of a subset of system calls, this factor does not necessarily translate into a higher overhead for monitoring.

6.3 Overhead for Fetching Arguments

Table 11 shows the overheads for accessing and/or modifying system call arguments. The overhead shown is the average taken across `ghostscript`, `tar` and `ftpd`. In Linux, accessing each system call argument requires a single system call in the monitoring process, so there is a modest overhead for getting these values. In IRIX, no system calls need be made to access the arguments, so there is no additional overhead. When we follow pointer arguments and access the monitored process memory, we have additional overheads. In most cases, we are interested in accessing arguments that are strings of relatively small size such as file names, and the overhead increase is small. But when we follow all pointer arguments, we end up reading the buffers used for file read and write operations, and this leads to a substantial overhead. In the case of Linux, there is an efficient way to perform bulk reads using the `mem` file in the `/proc` directory. But writing bulk data is very inefficient, as it requires one `ptrace` system call per 4 bytes of data to be written. Thus the overheads for modifying all arguments shoots up dramatically. This problem does not apply to IRIX since it provides efficient bulk read and write operations. Still, the overhead is high when we follow all pointer arguments, since this also implies that all system call entries and exits (including those for `read` and `write`) are being intercepted.

We conclude that the overheads are low when we limit ourselves to accesses to arguments such as file names. Intrusion detection, access control and confinement applications require accesses only to this kind of data, and hence the performance degradation introduced by these applications is low. Extensive access to data may be needed in some applications such as those for transparent data encryption or file replication. User-level interception introduces significant overheads for such applications.

7 Discussion

The runtime system described can be used for real-time monitoring of any application, in the absence of source code and without the need for changing or even recompiling any system software. Since our monitoring program resides in a separate address space, it is essentially nonbypassable. Having a user level approach provides us the ability to selectively monitor certain system calls, or monitor every call, thus controlling the granularity of control. These features, and a reasonable performance overhead give our system the power to add and enforce a variety of security policies, and we discuss some of the possible applications of our system below:

- *Policy-based Auditing:* In its simplest form, our system can record information about system calls and signals in a log file that can be used for archival purposes or as a source of information to investigate attacks. The information to be recorded can include system call arguments as well as system state related information, and can be customized on a per-process basis.

- *Intrusion detection:* As described earlier, a majority of intrusion detection techniques make use of system call information to perform their task. System call interception can enable them to obtain additional information that may not be available from audit logs, and thereby improve their ability to detect attacks. The ability to modify or abort system calls will enable intrusion detectors to take a more proactive role wherein they can prevent attacks from succeeding.

- *Access Control:* Fine-grained access control that is based on users as well as applications can be easily implemented using our infrastructure. Our user level approach has a distinct advantage in that users can deploy (or experiment with) new access control policies without any help from system administrators.
- *Confining uncooperative and malicious applications:* Since our interception and extension mechanisms cannot be bypassed, we can use them to confine uncooperative applications. Together with this factor, the mechanisms we provide for aborting system calls, modifying their arguments, and "locking" argument values (so that they cannot be changed with the intention of exploiting race conditions) give us the ability to confine malicious applications as well. Our ability to dynamically change the supervisor object (and hence the confinement policy) provides increased flexibility in confining malicious applications.
- *Other applications:* System call interposition has been used for other applications such as transparent encryption, data replication, etc. While our system can also support these applications, the performance overhead is higher. More efficient implementations, such as those based on shared wrapper libraries, offer a better alternative.

7.1 Drawbacks

One of the advantages of a kernel-based approach is that the extension code can perform operations in the context of the monitored program. In a user-level approach, we have to perform these operations in the context of the monitoring process, which is distinct from the monitoring process. For instance, operations such as changing the userid or file system root of the monitored process is not possible with our approach. Nevertheless, it is possible to constrain the monitored process to only files accessible to new userid or new file system root.

In a user-level implementation such as ours, the OS kernel treats the monitoring and monitored processes the same, and it is really up to the monitoring process to defend itself against a malicious monitored process. There are a number of ways in which the monitored process can try to cripple a monitored program. Not all of these may have been thought through completely by an implementor of the runtime system. Any "holes in the armor" resulting due to such oversight can be exploited by a malicious program to escape. Kernel-based implementations are harder to circumvent. The downside, however, is that errors in kernel-based implementations can cause significantly more damage than errors in the extension code executing at the user level.

Our implementation adds moderate overheads for monitoring, and this overhead is higher than that for kernel-based implementations. The silver lining is that most of the applications where the user is sensitive to performance are CPU-intensive, and for such programs, the overhead of our interception mechanisms are not even noticeable.

Our approach is implemented on top of the interfaces provided by the operating system features intended for debugging. Limitations in these mechanisms directly impact our infrastructure. For instance, in the current implementation of the `ptrace` mechanism in Linux, there is a race condition regarding the `fork` system call. In particular, the child process does not inherit the tracing flag, so it can run free until the monitoring process attaches to it. During this window of time, we do not have any ability to monitor or confine the child process. However, it appears that this race condition only arises in multiprocessor architectures. (This problem does not arise in SVR4 compatible systems, as they provide mechanisms that enable the monitoring program to arrange for the tracing flags to be inherited after a fork.) It appears possible to develop a work around by dynamic modification of code in the text segment, or better, modifying the instruction pointer to point back to a specially constructed sequence of instructions on the stack. The best solution, however, is to eliminate this "bug" from the OS implementation itself.

One of the assumptions of our approach is that the policy incorporated in the supervisor code is a "good" one, and will not itself cause damage or otherwise modify the capabilities of the monitored process in undesirable ways. The upside is that the scope of damage that can arise due to such errors is minimized by using appropriate high-level language for specifying the supervisor functionality, and also by having the supervisors run at user-level.

8 Related Work

There has been a lot of research done on improving the access control techniques available in operating system such as UNIX. Many of the works focussed on improving the filesystem protection model in UNIX [8, 19, 2]. Their implementation approach is based on modifying the operating system kernel to provide an enhanced protection model that can operate at a per-process and per-user level. The work of [28] develops a more powerful and flexible access control method called domain type enforcement (DTE). Using DTE, system administrators can specify fine-grained mandatory access controls over the interaction between processes and the objects accessed by them. This work is more closely related to ours in that their implementation is based on intercepting and checking system calls within the OS kernel.

Several research efforts have developed interception techniques that are implemented in libraries [16, 15]. While this approach leads to low overheads, they can be easily bypassed by code that directly invokes the system call without using the library, and are thus not appropriate for intrusion detection and confinement applications.

Interception of system calls within the kernel, coupled with the ability to add extension code that can be executed at this point, has been proposed by several researchers [9, 11, 20]. The approaches in [9, 20] use similar techniques for system call interception: they overwrite the system call table in the kernel (Linux or FreeBSD) to point to modules that will route the system calls through the extensions prior to and after the execution of normal system call functionality. Being kernel-based, these approaches generally have more flexibility and power in terms of what actions can be performed within the extension code.

A more comprehensive set of capabilities for extending operating system functionality was proposed in [11]. They allow for extensions to operate in user mode as well, but they run on top of the kernel level primitives developed by them, as opposed to making use of the features of commodity operating systems. Their work also deals with some issues not addressed by the works mentioned above, such as handling race conditions that might arise when a malicious multithreaded program attempts to change an argument of a system call (such as a filename) between the time it is checked by the extension code and the time the argument is actually used in a system call. Although it is difficult to provide absolute protection against such attacks using our approach, our technique of copying such arguments to a random location on the stack (beyond the stack pointer) provides a good solution. It would be very difficult for the malicious thread to search the entire stack (usually a few MB in size) to identify the location(s) into which the monitoring program may have copied the arguments.

The key distinction between our approach and that of the above works is that we aim for a user-level implementation of the system call interception infrastructure, similar to [13]. Their system is aimed at confining helper applications (such as those launched by web-browsers) to restrict their use of system calls. Due to their focus on "sand-boxing" of helper applications, their system provides abilities to only allow or deny a particular system call. Modifying system call arguments or the data referenced by them, or changes to the system call return code were not considered. Moreover, their work was based on Solaris and did not address Linux (which is our main interest) or issues of portability across other UNIX variants.

Several languages have been proposed to simplify writing system call extensions [9, 13, 26, 25]. Some approaches, such as [13] use a very simple, high-level language. Other approaches such as [9] are geared to maximize the capabilities that can be programmed into the extensions, and thus they use a superset of the C-language for writing extensions. Yet other approaches [26, 25] strive for a balance between expressive power, robustness of extensions and ease of writing extensions geared for a specific purpose such as intrusion detection/prevention.

9 Conclusions

In this paper we presented a new approach for developing a user-level infrastructure for system call interception and extension. Our work addresses several important issues not addressed by previous research. Even though we have a user-level implementation, it offers similar level of security and comparable level of capabilities as kernel-based implementations of system call extensions. This is achieved without the main drawbacks of the kernel-based approach, namely, normal users can develop and deploy their own extensions; moreover, damage due to errors in the extension code is limited, and does not bring down the entire system. As a result of this, our infrastructure can be used to develop extensions that accomplish a variety of security-related tasks such as custom auditing and logging, fine-grained access control, intrusion detection, confinement of uncooperative and malicious applications. Other applications such as fault-tolerance or encryption are also possible, but the performance degradation become significant for these applications.

Another important distinction of the research presented in this paper is that we have developed techniques to make our infrastructure portable across many versions of UNIX. More importantly, the extension code itself can be easily ported. Many of these techniques can be applied to in-kernel implementations of extension infrastructures as well.

We presented a comprehensive analysis of the performance impact due to user-level interception of system calls. The overheads are moderate in the worst case, and almost imperceptible under typical conditions. From a performance point of view, we believe that our results establish the practicality of user-level monitoring of system calls as a way to improve system security.

Our performance analysis, as well as the discussion in Section 7, indicate that the approach is well-suited for applications such as intrusion detection, access control and confinement of applications. Moreover, due to the limitations in OS-provided mechanisms for user-level interception of system calls, these techniques are currently better suited for confining uncooperative applications than malicious ones. It is also less suited for applications such as transparent data encryption or file replication that require extensive accesses to the memory of the monitored process.

References

[1] D. Anderson, T. Lunt, H. Javitz, A. Tamaru, and A. Valdes, Next-generation Intrusion Detection Expert System (NIDES): A Summary, SRI-CSL-95-07, SRI International, 1995.

[2] A. Berman, V. Bourassa and E. Selberg, TRON: Process-specific file protection for the UNIX operating system, USENIX Winter Technical Conference, 1995.

[3] M. Bishop, M. Dilger , Checking for Race Conditions in File Access. Computing Systems 9(2), 1996, pp. 131-152.

[4] T. Bowen et al, Operating System Support for Application-Specific Security, Personal communication, 1999.

[5] W.R. Cheswick, An evening with berferd, in which a cracker is lured, endured and studied, Winter USENIX Conference, 1992.

[6] Fred Cohen and Associates, The Deception Toolkit Home Page, `http://www.all.net/dtk/dtk.html`.

[7] D. Engler, M. Kaashoek and Jr J. O'Toole, Exokernel: An Operating System Architecture for Application-Level Resource Management, 15th ACM Symposium on Operating Systems Principles, December 1995.

[8] G. Fernandez and L. Allen, Extending the UNIX protection model with access control lists, USENIX Summer Conference, 1988.

[9] T. Fraser, L. Badger, M. Feldman Hardening, COTS software with Generic Software Wrappers, Symposium on Security and Privacy, 1999.

[10] S. Forrest, S. Hofmeyr and A. Somayaji, Computer Immunology, Comm. of ACM 40(10), 1997.

[11] D. Ghormley, D. Petrou, S. Rodrigues, and T. Anderson, SLIC: An Extensibility System for Commodity Operating Systems, USENIX Annual Technical Conference, 1998.

[12] A.K. Ghosh, A. Schwartzbard and M. Schatz, Learning Program Behavior Profiles for Intrusion Detection, 1st USENIX Workshop on Intrusion Detection and Network Monitoring, 1999.

[13] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer, A Secure Environment for Untrusted Helper Applications, USENIX Security Symposium, 1996.

[14] K. Ilgun, R. Kemmerer, and P. Porras, State Transition Analysis: A Rule-Based Intrusion Detection Approach, IEEE Transactions on Software Engineering, March 1995.

[15] M. Jones, Interposition Agents: Transparently Interposing User Code at the System Interface, 14th ACM Symposium on Operating Systems Principles, December 1993

[16] E. Krell and B. Krishnamurthy, COLA: Customized overlaying, USENIX Winter Conference, January 1992.

[17] C.Ko, M. Ruschitzka and K. Levitt, Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-Based Approach, IEEE Symposium on Security and Privacy, 1997.

[18] S. Kumar and E. Spafford, A Pattern-Matching Model for Intrusion Detection, Nat'l Computer Security Conference, 1994.

[19] N. Lai and T. Gray, Strengthening discretionary access controls to inhibit Trojan horses and computer viruses, USENIX Summer Conference, 1988.

[20] T. Mitchem, R. Lu, R. O'Brien, Using Kernel Hypervisors to Secure Applications, Annual Computer Security Application Conference, December 1997.

[21] T. Lunt et al, A Real-Time Intrusion Detection Expert System (IDES) - Final Report, SRI-CSL-92-05, SRI International, 1992.

[22] P. Porras and R. Kemmerer, Penetration State Transition Analysis: A Rule based Intrusion Detection Approach, Eighth Annual Computer Security Applications Conference, 1992.

[23] M. Russinovich and Z. Segall, Fault-Tolerance for Off-The-Shelf Applications and Hardware, Proceedings of the 25th International symposium on Fault-Tolerant computing, 1995

[24] R. Sekar, Y. Guang, T. Shanbhag and S. Verma, A High-Performance Network Intrusion Detection System, ACM Computer and Communication Security Conference, 1999.

[25] R. Sekar and P.Uppuluri, Synthesizing Fast Intrusion Prevention/Detection Systems from High-Level Specifications, To appear in USENIX Security Symposium, 1999.

[26] R. Sekar, T. Bowen and M. Segal, On Preventing Intrusions by Process Behavior Monitoring, USENIX Intrusion Detection Workshop, 1999.

[27] R. Wahbe, S. Lucco, T. Anderson and S. Graham, Efficient Software-Based Fault Isolation, 14th ACM Symposium on Operating Systems Principles, December 1993.

[28] Confining Root Programs with Domain Type Enforcement (DTE), K. Walker, D. Sterne, L. Badger, M. Petkac, D. Shermann and K. Ostendorp, USENIX UNIX Security Symposium, July 1996.