1 Introduction

One of the biggest problems faced by businesses, individuals and organizations is the protection of their networked computing system from malicious attacks launched remotely via the Internet. *Intrusion detection* refers to a broad range of techniques that have been developed over the past several years to protect against malicious attacks. Most intrusion detection techniques aimed at preventing intrusion 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. Therefore, if we monitor every system call made by every process, we may identify (and prevent) damage caused by an attack.

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, while system call interception enables offline techniques to access all the data needed for identifying intrusions, without incurring the overhead for accessing irrelevant information.

For the reasons mentioned above, we designed and implemented an infrastructure for active interception and modification of system calls. This report presents our infrastructure, which is a user-level system call interceptor like [2], where the system call performed by one process are intercepted (and possibly modified) by another process. Our approach has the following benefits:

- It provided an extensive set of capabilities for extension code;
- it developed an architecture and implementation that is easily ported to different versions of the UNIX operating system.

The rest of this report is organized as follows. Section 2 describes the design of our interceptor. Section 3 gives details about how to use the interceptor. In Section 5, we discuss the security issues. We show the performance results in Section 6.

2 Architecture and Program Components

2.1 Architecture

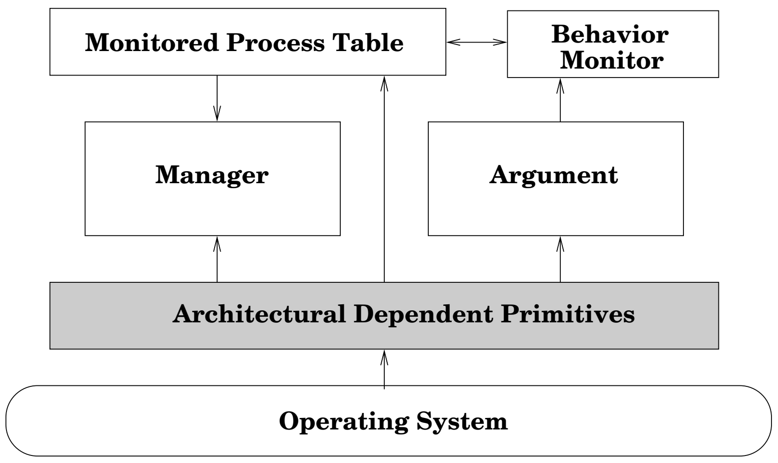

Figure 1: The architecture of interceptor (the arrows indicate the direction in which system call related information is flowed).

The architecture of the interceptor is shown in figure 1, which consists of five major parts, namely, *Architectural Dependent Primitives*, *Manager*, *Argument*, *Monitored Process Table* and *Behavior Monitor*. The Architectural Dependent Primitives (depicted by the gray rectangle in the figure) deal with details of the underlying operating system and interposition mechanism. They provide an easy-to-use interface to the rest of the

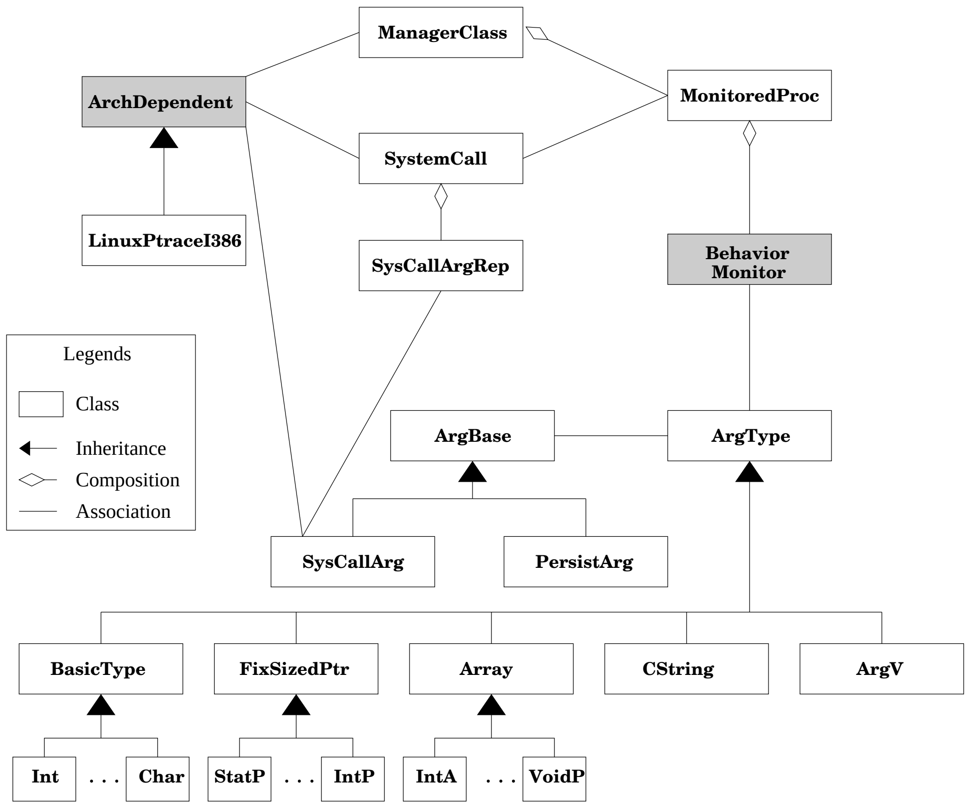

Figure 2: The static view of the interceptor

interceptor. Based on that interface, Argument deals with the representation details of system call arguments and provides a convenient interface to access them, as well as a caching mechanism to speed up accesses to used arguments. Manager is in charge of setting up the interception and receives all system call events from monitored programs, such as system call entries or exits. It also manages the states of the monitored processes, which is recorded in Monitored Process Table. After receiving a system call event, Manager updates the Monitored Process Table and then delivers the event to Behavior Monitor, which is user defined extensions for processing intercepted system call events.

As we have mentioned, one of the major goals of this interceptor is to make it easily portable. This is achieved by the design of Architectural Dependent Primitives, which is a high-level abstraction of the services that our interceptor needs from the underlying operating system and interposition mechanism. For example, we need a way to access a process's memory or register. The rest of the interceptor relies only on this set of interface, rather than the details of the operating system. Therefore, the only work needs to be done to migrate the interceptor to a new operating system is to rewrite the Architectural Dependent Primitives, according to details of the new system.

2.2 Program Components

In this section, we describe the static view of our interceptor, i.e., the classes building the interceptor and the relationships among them. We first show how the classes cooperate in a typical execution. Then, we examine the Architectural Dependent Primitives and Behavior Monitor in detail. Finally, to illustrate how the classes related to system call arguments are designed, we describe the process of system call argument logging.

2.2.1 Overview

Among these classes, `ManagerClass` is the control center. It controls the startup and termination of monitored programs, and delivers runtime events to their corresponding `BehaviorMonitors`. Right after the interceptor is started, the `ManagerClass` uses the `startTracing` method of `ArchDependent`, the class of Architectural Dependent Primitives, to put the monitored program under control. It then allocates a `MonitoredProc` object for the first process of the monitored program, and initializes required `BehaviorMonitor`s according to its configuration file.

After the monitored program begins execution, whenever it makes a system call, it will be stopped at both entry and exit point of the system call, and this system call event is reported to the `ManagerClass`. The `ManagerClass` first checks the type of the system call. If the system call indicates the creation of a new process, e.g. a `fork` with positive return value, the `ManagerClass` will allocate a new `MonitoredProc` for the new process and initialize its `BehaviorMonitor`. Otherwise, if the system call indicates a termination of a process, such as `exit`, the `ManagerClass` will free the `MonitoredProc` associated with the process. Then, the `ManagerClass` will deliver it to the process's `BehaviorMonitor`s for user processing.

The `BehaviorMonitor` receives system call notification from the `ManagerClass`. It can access system call and process related information, such as system call arguments and process id, through an interface, which we will describe in more detail in Section 2.2.2. After the processing of `BehaviorMonitor` is finished, the `ManagerClass` resumes the monitored program and waits for next system call event. It repeats this process until the monitored program exits.

2.2.2 Architectural Dependent Primitives

The Architectural Dependent Primitives are defined by the class `ArchDependent`. It consists of a set of abstract functions which is sufficient for our interceptor. These functions form a generalized interface, without dependency on particular operating system and/or interposition mechanism.

In order to make our interceptor work on an operating system, all we need is to implement a subclass of `ArchDependent` using mechanisms specific to the new operating system. Currently, as depicted in the figure, we implemented `LinuxPtraceI386`, an Architectural Dependent Primitives for Linux system on Intel X86 architecture using ptrace interposition mechanism.

2.2.3 Behavior Monitor

The `BehaviorMonitor` is an interface to let a user monitor the execution of monitored programs. The entry point to this interface is a virtual function called `deliverEvent`. The `ManagerClass` will call this function each time the corresponding process has a system call event intercepted. In `deliverEvent`, the user can access all information about the current system call and the current process. For example, `BehaviorMonitor` keeps a pointer to its `MonitoredProc`, which in turn can get a pointer to the `ArchDependent` object. Using the functions provided by the `ArchDependent` object, the user can get or change information of current process, such as user id (read only), effective user id (read only), content of register, contents of memory location, and etc. Through `MonitoredProc`'s pointer to `SystemCall` class, the user can also access and modify system call arguments, which we will describe in detail in the following section.

2.2.4 System Call Argument Logging

Using the functions provided by `ArchDependent`, a user can retrieve system arguments as a sequence of bytes, which is structureless and hard to use. Also, retrieving system call arguments data from a process's memory space is sometimes an expensive operation. For example, using Linux's ptrace interface, only four bytes can be retrieved in each system call. Therefore, we need an interface to access system call arguments conveniently and efficiently. That is the objective of system call argument related classes. In order to show a more understandable picture of how these classes works, we describe them through the process of system call arguments logging.

Before we start, let's have a overview of the related classes. `SysCallArgRep` is the class of system call argument representation. It keeps information of the location of a system call argument, either in memory or in register. It also caches the system call arguments for faster access. Using such a separate system call argument representation allows us to have *lazy access* to system call arguments, i.e., in a system call entry, we initialize the `SysCallArgRep` objects only with the location information; the access of system call argument data is delayed until necessary. The rest of classes are divided into two families, `ArgType` and `ArgBase`. `ArgBase` manipulates the byte streams of the system call argument data, while `ArgType` generates high-level view of the byte stream. `ArgBase` has two subclasses: `SysCallArg` interacts with the `ArchDependent` interface for system call argument access; `PersistArg` is a copy of the cached values of `SysCallArg` and is used to facilitate system call arguments logging. ArgType and its children classes are used to generate high-level and more understandable format of the system call argument data. For example, if we want to access a struct stat argument, we first use SysCallArg to get the byte stream from the process using information provided by SysCallArgRep. Then we use StatP class to format it into a struct stat format so that we can access it in a conventional way.

Now we will show how system call argument logging works. System call argument logging is built as a subclass of BehaviorMonitor. When a new system call event is delivered into it, the SysCallArgRep objects corresponding to current system call is retrieved through the pointer to current SystemCall object. Then, for each system call argument, a SysCallArg object is initialized with its SysCallArgRep, and its type is retrieved from the ArchDependent interface. Using the type information, we create a corresponding ArgType child object with the SysCallArg object. For example, if the system call is stat, we will initialize a StatP object with the SysCallArg object. Next, we will call the get function of the ArgType class to retrieve argument data from the process, which will initialize the data cache in the corresponding SysCallArgRep. A PersistArg object is then created with the cached value. Finally, after all the information of a system call is gathered, the system call is serialized into the log file. Later, when the log file is used, a user need to unserialize the system call one by one. In an unserialized system call, the system call arguments are represented as PersistArg objects. The user can then pass this object to its corresponding ArgType class, such as StatP, to get a high-level access of the argument's data.

2.3 Class Interfaces

Corresponding to each program component, there is a class in our implementation. In this section, we give a detail description of the important classes.

2.3.1 Class ArchDependent

The interface of class ArchDependent is as follows.

```
class ArchDependent
{
    public:
    virtual pid_t startTracing(const char *path, char **argv)=0;
    virtual int attachProc(pid_t pid)=0;
    virtual int detachProc(pid_t pid)=0;
    virtual int killProc(pid_t pid)=0;
    virtual int waitForCall(pid_t *newpid, SystemCall *call)=0;
    virtual int abortCall(pid_t pid, int rc, int errnum)=0;

    virtual long getReg(pid_t pid, int reg)=0;
    virtual int  setReg(pid_t pid, int reg, long val)=0;

    virtual int getData(pid_t pid, long addr, char* buf, size_t len)=0;
    virtual int setData(pid_t pid, long addr, const char *buf, size_t len)=0;
    virtual long allocMem(pid_t pid, size_t size)=0;

    virtual long getCallNum(pid_t pid)=0;
    virtual int  setCallNum(pid_t pid, long scno)=0;

    virtual int getReturnVal(pid_t pid, int *errnum)=0;
    virtual int setReturnVal(pid_t pid, int rc, int errnum)=0;

    virtual int getWorkingDir(pid_t pid, char *path, size_t size)=0;

    virtual int getExecPath(pid_t pid, char *path, size_t size)=0;

    virtual int getUID(pid_t pid)=0;
    virtual int getEUID(pid_t pid)=0;
    virtual int getGID(pid_t pid)=0;
    virtual int getEGID(pid_t pid)=0;
    virtual pid_t getPPID(pid_t pid)=0;

    virtual unsigned long getPC(pid_t pid)=0;
    virtual unsigned long getIP(pid_t pid)=0;
    virtual SysCallInfo *scInfo(int scno)=0;
} ;
```

Class `ArchDependent` defines the interface for one process to trace another, and to access the traced process's information. According to their functionality, the member functions are divided into two categories.

The first category is *process control*. These functions are used to control the execution of the traced process. `startTracing` takes a vector of command line arguments, and starts tracing the program specified by the command line. `attachProc` is another way to trace a process, which uses process ID of an existing process, instead of command lines. `detachProc` detaches a traced process, making it running without tracing. `killProc` terminates a traced process. `waitForCall` is the most important function in `ArchDependent`, which reports a new system call event. It return the process ID of the new event by the argument `newpid`. Its second argument `call` contains information about a system call, which will be described in Section 2.3.4.

The second category is *data access*. This category includes functions used to retrieve process execution related data, such as the register contents and the effective user ID. `getReg` and `setReg` are used retrieve and change the contents of the process's register, respectively. The register is specified by the argument `reg`, whose values is specified by the header file `<sys/reg.h>`. `getData` retrieves `len` bytes memory contents starting from `addr` from the traced process into `buf`, while `setData` write the `buf` into the process's memory. The next four functions deal with the system call number and return value of current system event, respectively. `getWorkingDir` retrieves current working directory of process `pid`, and `getExecPath` retrieves the executable file of process `pid`. `getUID, getEUID, getGID, getEGID` and `getPPID` returns the user ID, effective user ID, group ID, effective group ID, and parent's process ID, respectively. `getPC` returns the location from which the current system call is called, and `getIP` returns the current program counter. The last call of `ArchDependent` is `scInfo`, which returns a structure containing information of the system call specified by `scno`, such as system call name and the type of each argument.

2.3.2 Class ManagerClass

```
class ManagerClass
{
    private:
    ArchDependent *arch;
    ProcHashTable<MonitoredProc> monitoringTable;

    public:
    ManagerClass(ArchDependent *pAD, const char *mapfile);
    ~ManagerClass();

    int startTracing(char* argv[]);
    int startTracing(pid_t pid);
    ArchDependent *getArch();
};
```

Class `ManagerClass` has maintains a pointer to the current `ArchDependent` object, and a hash table `MonitoredProc`, which stores information of traced processes. To trace a process, a program needs to create a `ManagerClass` object with a pointer to current `ArchDependent` object and the location of mapping file (see Section 3.1). It then can start the program by call the `startTracing` member function, either by providing the full command line argument vector, or by providing the process ID of the program to trace. Function `getArch` return the current `ArchDependent` object for the rest part of the interceptor to access process related information.

2.3.3 Structure SysCallArgRep

```
struct SysCallArgRep
{
    bool ISreg_;
    long addr_ ;
    bool inited_;
    long data_;
    char *buf_;
    size_t size_;
};
```

Structure `SysCallArgRep` holds the representation of a system call arguments. The following assumptions are made about system call arguments: (a) they can be stored in a long; data that wont fit within a long is stored in memory, and a pointer to this buffer is the actual system call argument; (b) they are contained in a register or memory.

Field `ISreg_` specifies whether this argument is a register. `addr_` records the location of the argument. For a register argument, it is the register number as in `<sys/reg.h>`; for a memory argument, it is the address in memory. Field `data_` is a local cache to store system call arguments the is in monitored process's register or memory space, and `inited_` indicates whether the cache is valid. If the argument is a pointer to another structure or string, `buf_` is used to cache its contents. For example, in `open` system call, the first argument is a string specifying the file location. After the file name is retrieved, it is cached in `buf_`. The `size_` records the size of cached `buf_`. If the argument is not a pointer, or the pointed value is not used, `buf_` is `NULL` and `size_` has a zero value.

2.3.4 Class SystemCall

```
class SystemCall
{
    private:
    long uscno_ ;
    bool isEntry_ ;
    SysCallArgRep *arg ;

    public:
    SystemCall();
    ~SystemCall();

    long uscno();
    bool isEntry();
    SysCallArgRep *getArgRep(int number);
} ;
```

Class `SystemCall` records the information of current system call event. `uscno` reports the current system call number, and `isEntry` tells whether the current event is a system call entry or system call exit. System call arguments can be accessed by `getArgRep`. It return the pointer to the argument's corresponding `SysCallArgRep` object. In the next two sections, we will see how this object is used to retrieve system call arguments.

2.3.5 Class ArgBase

```
class ArgBase
{
    public:
    virtual long getData()=0;
    virtual int  setData(long dat)=0;
    virtual void* getBuf(size_t size)=0;
    virtual void* getDelimBuf(size_t size, unsigned char delim)=0;
    virtual char* getStr(size_t size)=0;
    virtual bool setBuf(const char *buf, size_t size)=0;
    virtual bool setNewBuf(const char *buf, size_t size)=0;
} ;
```

Class `ArgBase` provides an abstract interface for accessing system call arguments. We explain its interface by describing how it child class, `SysCallArg`, handles system call arguments. In `SysCallArg`, `getData` initializes the `data_` field in the `SysCallArgRep` from either register or memory, while `setData` write the value of `dat` into the argument's register or memory. If the arguments is a pointer, `getBuf` will retrieve a block of size `size` from where the pointer points to. `getDelimBuf` is similar to `getBuf`, but it only retrieves the buffer before a delimiter `delim`. A special case of this function is `getStr`. It retrieves a string pointed by the argument, in which case the delimiter is the value zero. Those functions will be used by the class `ArgBase` to provide a higher level view of the system call arguments.

2.3.6 Class BehaviorMonitor

```
class BehaviorMonitor
{
    protected:
    MonitoredProc* mp;
    public:
    BehaviorMonitor(MonitoredProc* MP);

    virtual int deliverEvent();
    virtual BehaviorMonitor* clone();

    SysCallArgRep *getArgRep(int i) ;

    String programName();
    String getCwd();

    int getPID();
    int getPPID();

    int getUID();
    int getEUID();

    int getGID();
    int getEGID();

    ...

};
```

Class `BehaviorMonitor` is the interface of our interceptor and the user supplied behavior monitor module. Among the member function, the `deliverEvent` is the entry point. Upon receiving a system call event, `ManagerClass` will call this function of the `BehaviorMonitor`s of the system call's process. Therefore, users need to put their monitoring logic into this function. Other functions are used to provide process information to the user's behavior monitor. For example, the function `getPID` returns the process ID of the system call event's caller.

3 Tutorial

This section shows how to assign behavior monitors to a program, and how to start monitoring a program using our interceptor.

3.1 Mapping file specifications

Our interceptor uses mapping files to associate behavior monitors to programs. The mapping file is a text file containing lines of rules. Each line is in the following format(lines that begin with # is considered to be comments):

```
PROGRAM    LOCATION   CLASS-NAME
```

- PROGRAM is the canonicalized absolute program path name, i.e., absolute path names with all symbolic links expanded and `.` or `..` resolved. The word “default” is used to match all path names not specified in a mapping file.
- LOCATION is the path name of dynamic library containing the specified behavior monitors.
- CLASS-NAME is the class name of the behavior monitors to be associated with the PROGRAM. It’s case sensitive. Two behavior monitors are predefined, `KILL` and `NONE`. `KILL`’s behavior is to terminate the monitored process while `NONE` does nothing about the events. To use these two predefined extensions, the LOCATION field must have the value “PREDEFINED”.

Behavior monitors are organized in a layered fashion by the order in which they appear in the mapping file. The earlier an behavior monitor of a program appears in the mapping file, the closer it is to the program, i.e., when an entry of a system call event is delivered, the earlier it receives the event. After the kernel finishes the system call, the exit of the system call is delivered in the reverse order, i.e., the closest behavior monitor to the program gets the event delivered last. That’s how the behavior monitor layer works.

3.2 Tracing programs using the interceptor

After the behavior monitor is specified by a mapping file, the interceptor can be started with the following command:

```
tracer -f mapfile [ -p pid | command ]
```

The mapping file is passed to the interceptor by the -f option and there are two ways to specify a program. One way is to pass the whole command to interceptor after specifying the mapping file; if the program is already running, interceptor can be attached to the program by specify the pid using -p option. In the second case, the executable name is retrieved from the system to look up behavior monitors in the mapping file.

4 Guide for developers

4.1 Migrating UI to another operating system

In previous sections we saw the various components of the system. In this section we will go through the components in detail so as to aid a programmer in porting the tracer (interceptor) across platforms. In the first section we will cover some design decisions that were made during the design of the interceptor. In section 3.2 we will cover the Architecture Dependent class that has to implemented in the target platform. and in section 3.3 the Architecture Independent classes which can be used without any change across platforms but there is a need to understand these classes too as they call on the functions in the Architecture Dependent class.

4.2 Assumptions

The design of the tracer is not specific to any operating system and tracing mechanisms. Instead, we defined a set of abstracted interfaces, describing what the operating system and tracing mechanism needs to provide. As long as those interfaces can be implemented on a particular system, our tracer can be supported. And it is the only work needs to be done.

Our tracer is based on the following assumptions.

- Each process/thread can be identified by a handler, which can fit into the size of a long word.

- A process/thread access the operating system services through system calls. Each system call can be identified by a numerical identifier.
- All the arguments of a system call can fit into a long word. That means, an argument larger than a long word must be passed by pointers. The arguments are either stored in registers or in memory cells.
- The tracing mechanism to be used needs to intercept a system call before and after the system call's execution. Once the system call is intercepted, the corresponding process/thread is stopped until the tracer finishes processing.
- The tracing mechanism need to provide ways to access the memory and registers of a traced process/thread.
- The tracing mechanism *must* be able to trace a new-spawned process/thread.

4.3 Architecture Dependent Components

This is the abstract interfaces describing the needs from an operating system and tracing mechanism. According to the functionalities, they are classified into four categories, namely, *data type mapping*, *dynamic library support*, *tracing support*, and *data access*.

4.3.1 Data Type Mapping

These are some typedefs which are needed to ensure cross platform porting.

NAME	LINUX EQUIVALENT
PID_TYPE	pid_t
PATH_MAX	PATH_MAX
SIGNALS_MAX	SIG_MAX

4.3.2 Dynamic Library Support

This section deals with the need for handling Dynamic Libraries. To setup extensions the tracer needs to load it dynamically this way we don't have to tie the extensions at compile time.

a) Function openDynamicLibrary Input : path Output : handle Description: Opens the specified dynamic library and returns a handle to it.

b) Function getDLFunction : Input : handle, symbol name Output : address Description: getDLFunction takes a "handle" of a dynamic library returned by dopen_library and the null terminated symbol name, returning the address where that symbol is loaded.

c) Function closeDynamicLibrary Input : handle Output : Success Description: closeDynamicLibrary is called after all operations on the dynamic library have been performed. If no other process is using this library then the operating system usually unloads it.

d) Function getDLError Input : None Output : Error Description : If in any of the above functions there was some error, getDLError returns that error.

4.3.3 Tracing Support

These functions do the setup for the interception and the actual interception. The also return information back to the control classes (Manager class for example) based on which decisions can be taken.

a) Function startTracing : Input : path, arguments Output : Process id Description: Creates a new process for the program located in 'path' with the 'arguments' supplied. Returns Process id of the created process. Function also updates information that this process is being traced.

b) Function attachProc : Input : Process id Output : Success Description: Starts tracing a process given by the Process id. Returns success. Function also updates information that this process is being traced.

c) Function detachProc : Input : Process id Output : Success Description: Removes the process from the control of the tracer. It also updates information that this process is no longer being traced.

d) Function killProc : Input : Process id Output : Success Description: Kills the process. It also updates information that the process has terminated hence is no longer being traced.

e) Function waitForCall : Input : Process id, System Call Output : Status of the next call Description: In this function, the program waits for the next system call and finds the corresponding control block.

f) Function abortCall : Input : Process id, return val, error_num Output : Success Description: Stop the system call from executing/terminate the system call.

4.3.4 Data Access

These functions are used to get and set data from the Operating System.

a) Function getReg : Input : Process id , register number Output : Content of specified register. Description: Get the contents of the specified register.

b) Function setReg : Input : Process id, register, input value Output : Success Description: Set the contents of the specified register with the input value.

c) Function getData: Input : Process id, addr, buf, len, buffermode, size Output : Success Description: This function is used get argument data for arguments which are stored in memory. Get the Data of size 'len' bytes from addr into 'buf'.

d) Function setData : Input : Process id, addr, input string, len Output : Success Description: This function is used set argument data for arguments which are stored in memory. Function writes 'len' bytes of data from 'input string' to 'addr'.

e) Function getUID Input : Process id Output : UID Description: Get the UID for the process.

f) Function getEUID : Input : Process id Output : EUID Description: Get the EUID for the process.

g) Function getGID : Input : Process id Output : GID Description: Get the GID for the process.

h) Function getEGID : Input : Process id Output : EGID Description: Get EGID for the process.

i) Function getPPID : Input : Process id Output : PPID Description: Get PPID for the process

j) Function allocMem : Input : Process id, size Output : address Description: Allocate random memory on the stack and return the address.

k) Function getCallNum : Input : Process id Output : System call number Description: Get the number of the current system call for the process being traced.

l) Function setCallNum : Input : Process id, System call number Output : Success Description: Set the number of the current system call for the process being traced.

m) Function getReturnVal : Input : Process id, error_num Output : return value of system call Description: Get the return value of the current system call and if return value is -ve then error_num stores the value.

n) Function setReturnVal : Input : Process id, retval, error_num Output : Success Description: Set the return value of the current system call and if retval is -ve the error number in error_num is used.

o) Function getPC : Input : Process id, system call number, indirect call, stack length, number of frames, pointer to call stack, initial offset, offset, useoffset Output : PC Description: Gets the program counter at the point in the program where the system call was made. Note that the system call could be made by the code directly or by a function in a library called by the code. The getPC returns the program counter at the time of making the system call in the code or the program counter value when the code called the library function.

5 Security Issues

5.1 Race condition

In [1], the author discussed several types of race conditions, namely, symbolic link race, relative path race, argument race, file system information race and shared descriptor space races.

For symbolic link race, relative path race and file system information race, they make use of the ambiguity of the path name argument and make changes after the time of check and before the time of use. To prevent these race conditions, our solution is to replace the ambiguous name with the canonicalized absolute path, which uniquely identifies the file in the file system.

For argument race, our solution is to move the argument to the process's private memory space, the top of process's stack. It is still vulnerable if there are threads in the process's space, because the other threads can scan the stack and possibly change the argument value. To defend against this possibility, we adopt the method of [3], putting the arguments at a randomized address of the stack.

The shared descriptor space race is currently not handled.

Program	Total Calls	Call frequency	Normal time	Traced time	Overhead
gzip	17289	446/sec	38.731s	38.961s	0.6%
gostscript	10739	14357/sec	0.748s	0.897s	19%
tar	87319	85106/sec	1.026s	1.830s	78%
ftpd	12629	145161/sec	0.087s	0.174s	100%
xpdf	48915	211753/sec	0.231s	0.572s	148%
ls	19733	243617/sec	0.081s	0.223s	173%

Figure 3: Performance Result and System Call Frequency Information

5.2 Guard the monitoring process

Because the monitored process and monitoring process run with the same user, it is possible that the monitored process can affect the execution of monitoring process, such as sending a signal. Fortunately, it must be done by a system call. Therefore, after such system calls are intercepted, we check whether they are targeted to the monitoring process. If so, the call will be aborted. For example, if the pid argument of `kill` system call is the pid of monitoring process, the signal argument will be changed to 0, which does no harm to the monitoring process.

6 Performance Evaluation

To evaluate the performance impact of the tracer, we tested it using several commonly used UNIX applications, namely, `gzip, gostscript, tar, ftpd, xpdf` and `ls`. The test was done on a Dell Inspiron 4150 laptop, which has a Mobile Intel Pentium 4 1.80GHz CPU with 512KB cache, 512MB memory and a 40G 5400RPM hard disk. During the test, the tracer intercepted all the system calls made by the tested process and its children, but no more operations were performed unless they were necessary, such as attaching to the new child process after it was created. The testing data used for each application is described as follows.

- `gzip:` A data file of 160MB was used as the input to `gzip`, and the output file was 60MB.
- `gostscript:` In the test, `gs` was used to interpret an academic paper which contained 10 pages and whose size was 167KB.
- `tar:` The `/usr/bin` directory of a Redhat Linux 8.0 system was used to generate the archive. It contained 2173 files and the size of the output file was 160MB.
- `ftpd:` We transfered a data file of 10MB using the wu-ftpd server. To avoid the interference of disk and network I/O, the server was setup on localhost and the output of the client was redirected to `/dev/null`.
- `xpdf:` An 180KB academic paper which consisted of 10 pages was interpreted during the test.
- `ls:` A directory structure containing 3139 entries was used to test the performance of `ls`. `ls` was invoked with the `-lR` command line option, which is to display long list format of each file and to list the subdirectories recursively.

Figure 3 shows the results of the performance test. In this table, *total calls* is the total system calls made by the process and all its children. *Normal time* is the sum of user times and system times of the tested process and all its waited children during normal execution. *Traced time* is the sum of user times and system times of those processes running under the monitor of the tracer. The *call frequency* is calculated by dividing the "total calls" by the "normal time". From the above results, we can see that computation intensive programs like `gzip` have less overhead while system call intensive programs like `ls -lR` suffer from more performance penalty. To further explore the relation between the system call frequency and the performance overhead, we plotted the data of Figure 3, which is shown in Figure 4. We can see that the overhead is proportional to the system call frequency.

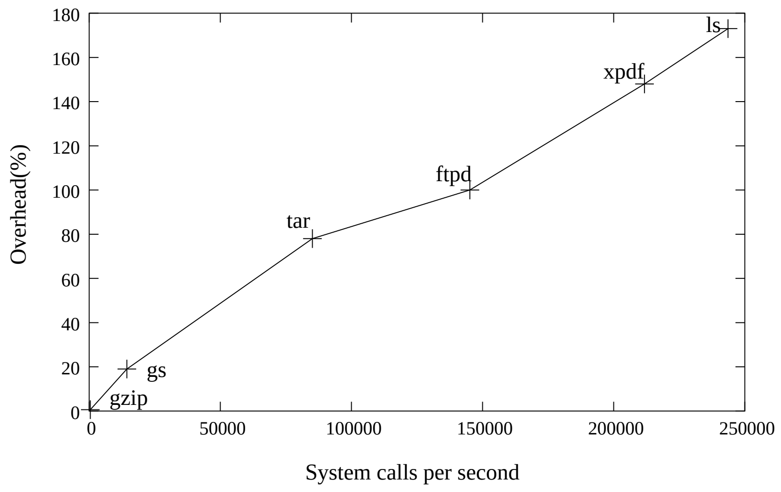

Figure 4: Relationship between System Call Frequency and Overhead

References

[1] T. Garfinkel. Traps and pitfalls: Practical problems in in system call interposition based security tools. In *NDSS*, 2003.

[2] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications: confining the wily hacker. In *USENIX Security Symposium*, 1996.

[3] K. Jain and R. Sekar. User-level infrastructure for system call int erposition: A platform for intrusion detection and confinement. In *ISOC Network and Distributed System Security*, 2000.