To be decided
— Individual Research Project —

Giuseppe Pes
giuseppe.pes12@doc.ic.ac.uk
Supervisor:Prof. Cristian Cadar
Course: XXXX, Imperial College London

April 10, 2013

Abstract

MY ABSTRACT

Acknowledgements

I would like to acknowledge ...

Contents

List of Figures

Listings

Chapter 1

Introduction

1.1 System Call interceptor

Regardless the nature of an application, resources such us files, socket or shared memory can be accessed only via the system-call interface exposed by the operating system. The sequence of system calls invoked by an application fully characterised its behaviour, that thus can be monitored and regulated.

*A **system call interceptor** is a powerful technique for investigating and regulating all application's interactions with the operating system via system call invocation. It usually works by interposing a third agent,called monitor,between the operating system and the application of interest. The interposition agent is usually notified before the system call is executed. It should be able to analyse the system call arguments and decide whether the execution of a system call should be completed or not.*

Interposition can be used to provide programming facilities that are usually not offered by an operating systems. Some areas where an interposition technique is employed include :

Monitoring System call tracing and monitor facilities such as strace [21] use a system call interceptor for monitoring the program's use of system services.

Software confinement There has been a lot of research to improve a system call interception in terms of security and performance as it is the base mechanism for security tools such as sandboxes and introduction detection[ref].The resource control provides by an operating system is usually based on a Discretionary Access Control DAC (Linux, UNIX-like system) model, where all programs executed by a user inherit his permission for accessing resources. This model is simple and quite effective, though it does not provided any protection against malicious code (back-door) and flaws (buffer overflow). Sandboxing software [2, 43, **?**, 44, 49] have been developed to overcome these shortcomings. A sandbox provides a fine-grained control for the resource access by the sandboxed program. These software use a system call interceptor for monitoring all system calls made by the sandboxed program and check them against a policies file which constrains a program to a correct behaviour. If one of this policy check fails the system call is not executed and a possible malicious action is prevent.

Introduction detection Introduction detection via analysis of system call traces has received attention throughout recently years [23, 20]. The problem of a introduction detection tool is similar to that of a sandboxing tool previously described.Though in this case the monitoring process is only interested in analysing what calls an application makes in order to find out anomalous system call sequences that may identify an introduction in the system

Debugging A debugger as GDB uses an interposition technique to catch some or all of the system calls issued by the application being debugged, and show the related information for each system call.

Portable environment Tools such as CDE [18] whose aim is to ease the pain of software deployment. CDE eliminates the dependency problems which raise compiling and installing a software in a different environment from that in which the application has been developed. It creates automatically portable software package using a system call interceptor to track the execution of x86-Linux and collect all data,code and environment required to run it in another Linux machine.

Virtualisation A system call interceptor has been successfully used to fully virtualise the system calls made by a program in [9, 48, 8]. UML is a virtualisation technology which enables multiple Linux systems to run as an user application within a normal Linux system. The system call interceptor is used to redirect all system calls made by a program to the guest kernel. UML is primarily used for kernel debugging and sandboxing purposes.

Multi Variant Execution Multi-variant code execution [38, 5] is a run time monitor technique that finds out and prevents malicious code from executing. The idea behind this method is to run two or more slightly different version of the same application in lockstep.At certain synchronization point their behaviour is compared against each other and if a divergence is found, a notification is raised. In multi variant execution,the invocation of a system call is a synchronization point as all variants must make exactly the same system calls with the same arguments within a temporal window. In order to determine if the variants are synchronized with each other the monitor process intercepts all system calls invoked by all variants and compares their arguments.

Record Replay Record-Replay tools such us Jokey [37] are based on the observation that the system call sequence invoked by a program fully characterized its behaviour. They log the execution of an ordinary program and replay deterministically it later. These tools provide a way to reproduce anomalous behaviour or crashes in controlled application allowing for a developer to find out the cause of a bug.

As can be seen from the previous list the applicability area of a system call interceptor is rather broad spreading from debugging applications to security enhancements and virtualisation. The aim of this document is to present a detailed analysis of Linux for the purpose of building interposition tools. Next paragraph surveys different techniques that may be employed in developing a system call interceptor in a general fashion. Then these techniques are recalled and extended throughout the rest of this document. The last two sections of this chapter present requirements and limitations of a system call interceptor.

1.2 State of art in system call interposition

Each context on which a system call interceptor is used has its own constrains and requirements. For example in a sandbox application the main requirement is the security, while for a record-replay toll security might not be as important as a low overhead tracing mechanism. Therefore, different ways to implement a system call interceptor have been developed to satisfy different requirements. Some of the most widely used approach to implement a system call interceptor in Linux are :

Ad hoc modified library A modified library for the purpose of intercepting the system call made by the program to which it is linked, was the first approach used for implementing a system call interceptor [40, 37]. It relies on the fact that system call are accessed via wrapper functions, within the glibc library. A system call interceptor can thus be realised by linking the application of interest to a different library which contains an instrumented version of the wrapper functions. This approach has been implemented in [40]. Its major benefit is that its performance are almost the same as no instrumented application, but it can be easily bypassed invoking directly a system call using low level mechanism.

Trace features The kernel provides features such as ptrace and utrace for tracing and debugging programs. Although this features has been design for debugging purpose, they can be used to

successfully build up a system call interceptor in user space without kernel modification (Utrace needs to write a module).

Binary rewriting Binary rewriting facilitates the insertion of addition code[1] into an binary executable file in order to monitor or modify its execution. A binary file thus can be modified in a such way that allow for a additional code to be executed before and after a system call providing a a way to observe and regulate them.This techniques can be applied both statically as well as dynamically and can be done in a number of different ways, e.g. full binary rewriting where the entire binary is rewritten [**?**, **?**] or selectively rewriting where only sensitive instructions (i.e system call instructions) are rewritten.

Seccomp mode-based approach Seccomp short for *secure computing mode* is a simple sand-boxing mechanism provided by the Libux kernel. This secure environment can be activated by issuing a request via `prtcl()`. Currently Linux supports two different version of seccomp. The first introduced in 2005, allows a process to make only 4 system calls `exit()`, `sigreturn()`, `read()` and `write()`. If the process attempts to call a different system call from the previous ones, the kernel will terminate that process via SIGKILL signal. It has been used to implemented two sandboxes [**?**, **?**]. The second version called seccomp-bpf has been introduced in the 3.5 kernel as enhancement of the previous version. This works by processing each system call request through a BPF filter within the kernel. Seccomp-bpf has been employed to increase the security of software such *Chrome*, *Chrome OS* and *vstpf*.

Kernel enhancement In a kernel based implementation, the system call interceptor is implemented within the operating kernel, and all the extension code is executed in kernel mode. A system call interceptor may be inserted modifying the source kernel [49] or it can be uploaded via a kernel module [43]. Both solutions offer the same power in terms what can be accomplished within the extension code, though the later does require neither to compile nor to patch the kernel, it is thus more portable. The kernel approach is exposed in more details in 3.

1.3 Requirements

So far we have provided a general introduction to all possible mechanisms Linux provided to implement a complete system call interceptor, further all of them will be extended explening how a system call interceptor can be implemented using it, providing real example in which such mechanism has been used. Which features are needed by a system call interceptor? A system call interceptor requires a minimum number of capabilities to be effective and thus providing a good base which a sandbox can rely on:

Monitor capacity A system call interceptor must intercept all attempts to invoke a system call made by the untrusted process before that the system call is executed by the kernel. Furthermore, it also needs to provide a method to analysis the arguments of the system call (and to access to the applications data space if the real argument is located there) and returned values. This requirement is the core of the intercepting mechanism itself.

Fine-granaid control we should be able to specify which system call should be intercept and which should be not. Regardless the method used to implement it; a system call interceptor always introduces an overhead with respect to the normal system calls flow. This greatly improves the performance because we could, for instance, trace the system call which gain access to a new resource but avoid to trace those that use the resource already open.

Preventing the system call execution when a system call is invoked with unsafe parameters such as open(/etc/passwd);. The system call interceptor must have a means of aborting its execution without aborting the entire process and setting a proper return value i.g. EPERM.

[1]The additional code is often referred as *instrumentation code*

Monitoring all children The system call interceptor must intercept and monitor all children of the traced processor. It is crucial that the children of a sandboxed process must be constrained to the father's policy rules.

dealing with multithread application another important requirement it the possibility to trace all threads of the traced application efficiently, without stopping them if not necessary.

Why characteristics should a System call interceptor have?

1.4 Issues

conflicts parameters

1.5 Design goal

efficacy A good system call interceptor must ensure that all system call are correctly intercepted and traced. Furthermore, this rule must be also valid for all process spawn by the process currently monitored.

efficiency Performance is a crucial aspect of a system call interceptor as the tracing mechanism introduces a remarkable over head reducing the application's performance.

flexibility An intercepting mechanism should be enough flexible to implement a large range of features. For example, it should support the ability to access and rewrite arguments of a system call, to change a call's return value or to change the privilege level of a process while it executes a system call.

compatibility A system call interceptor must be compatible with a wide range of software. It must not require applications to be recompiled or modified in a different way by a user in order to catch their system call.

versatility A user should be able to configure which system call must be intercepted and which not. This is an important feature as it reduce the overhead due to the tracing mechanism.

deployability An intercepting mechanism should be easily portable among different platforms. This can be achieved reducing the dependencies to library or kernel features and easing the installation process.

Chapter 2

Ptrace

Like most of the UNIX-like operating systems, Linux supports the **ptrace** system-call interface. Ptrace provides a means by which a process might observe and control the execution of another process; the first process is referred to as the *tracer* process and the latter as the *tracee* process. Since its introduction in the Linux kernel version 1.0, ptrace has been mainly employed in implementing debuggers, such as GDB and DBX, where it is used to insert breakpoints into the running program; and for system-call tracing. For example, the command strace exploits ptrace requests to retrieve information regarding the system-calls invoked by a program.

The ptrace's interface is:

```
long ptrace ( enum __ptrace_request request, pid_t pid, void *addr, void *data
    )
```

Listing 2.1: Synopsis ptrace system call

Ptrace supports numerous different type of actions, for a comprehensive list see [27], though the most representative ones are:

- Attach to, or detach from the process being traced (the tracee), used to install the tracing mechanism.
- Read and write the tracee's memory and its status registers.
- Signal injection and suppression.
- Resume the execution of the tracee allowing it to continue until an event of interest occurs (e.g. a system-call is invoked).

The tracer process gains extensive control over the operations of the tracee process. This includes manipulation of its file descriptors, memory, and registers. It can single-step through the tracee's code, observe and intercept system calls and their results. Furthermore, it can manipulate the tracee's signal handlers and both receive and send signals on its behalf. While a process is being traced, all events such as attempting to invoke a system call or receiving a signal are turned into a `SIGCHLD` [1] signals that are delivered to the tracer process. Every time the tracee receives a signal, it is stopped so that the tracer may analyses its status and if necessary changes its execution.

Although ptrace has been primarily designed for debugging purposes, it offers enough flexibility to be used for different tasks as well. It has been successfully and widely used to implement a *user-space system call interceptor* in [REFERENCES], though it entails numerous limitations. The major disadvantage of the ptrace approach is that the performance may be remarkably reduced in a program which uses intensively system-calls. Several context switches between kernel space and user space are introduced respect to a non-traced execution to support the tracing mechanism. However, an

[1] The communication between the tracee and the tracer is performed using the standard UNIX parent/child signaling over *waitpid* system call

interceptor mechanism can be employed in different contexts and therefore different criteria can be used for choosing it. For example, when performance is not a fundamental requirement such as for debugging, ptrace offers a valid solution.

One of the most important feature offered by a system-call interceptor is the possibility of retrieving the system call number and its arguments. This may be used in applications for auditing purposes such as strace [ref to strace], where all system calls and their arguments are recorded or sandbox application [references] where each system call is tested against a policy specifying which arguments are allowed and which are not.

Ptrace provides the possibility to retrieve both system call number and its parameters by accessing the status registers and the space memory of the tracee. Accessing the tracee's memory via ptrace is actually rather inefficient as ptrace permits only to pass a small fixed-size block between the two process. This is one the crucial limitations of ptrace and different techniques have been developed during the recent years to mitigate it. These are presented in the 2.3 section.

Another requirement missed by ptrace is to support multi-thread applications. Ptrace actions are specified for single thread via its ID, thus in a multithread application each of its threads must be attached individually. This action if not handled carefully might leave open a temporal window during which some system-calls might not be intercepted. The section 2.4 surveys different problems regarding tracing multithread application and provides an effective solution to them. An important requirement for an application such as a sandbox is the possibility to deny the execution of a system call when it does not satisfy the policy. While other operating system, such Solaris provide a way of aborting a system call, ptrace does not offer any. In the section 2.5 different methods to overcome this shortcoming are introduced.Finally, another requirement missed by ptrace is the possibility to trace only a subset of the system calls provided by the operating system and leave others to be executed without any overhead. This reduces the tracing overhead in several different cases, for example, in a sandbox application where only the system calls which might affect the security of the operating system should be intercepted. In [49] all system calls which open or change a file descriptor (open, socket, bind) are intercepted, while those which use it (read, lseek, write) are not.

Ptrace has been used also for virtualising system calls in [9, 48, 8]. In this case all system-calls invoked by the tracee must be nullified in the host architecture in order to be virtualized; see section 2.5.The ability to write into the tracee's memory allows the tracer to change also the tracee own code segment. This feature has been exploited by specialised programs [reference] to patch running programs in order to avoid bugs or to fix security lacks.

Although its limitations and shortcoming ptrace is the standard tracing mechanism used in Linux and Unix-like operating system. This success is due principally to two factors. The first is that ptrace provides an easy way to set up the tracing mechanism in user space compared with other approach such us kernel-based tracing mechanism. It can be used without root privilege, provided the tracer has enough privilege to trace the process of interest. Secondly, even though ptrace is not a POSIX system call, it allows for the interception infrastructure to be easily ported between different operating systems.

The remainder of the chapter introduces how to set up a system call interception using ptrace section 2.1 and how to overcome the shortcomings and limitations discussed before.

2.1 Ptrace tracing mechanism

Ptrace may be successfully used to build up an effective system call interceptor, ensuring that the tracer process intercepts all system call made by the tracee process and its child processes. This section exposes in details how to implement a system call interceptor using ptrace.

There are two different ways to install the tracing mechanism. The tracer might invoke the fork system call to create a child process which notifies the kernel its willingness to be traced by calling ptrace with the argument `PTRACE_TRACEME`. This request causes the kernel to set a trace flag (`PT_TRACED`) in the target process descriptor and stop it by setting its state to non-interruptible sleep. This approach is usually used before the tracing software initiates a new program invoking an exec system call. Alternatively, the monitor process can make a request to the kernel for tracing a running process

via ptrace specifying the argument PTRACE_ATTACH and the target process pid. In this case, there are three conditions to be checked:

1. If the privileges of the tracer process allow it to trace the tracee process. Ptrace can attach only to processes that the owner can send signals to (typically only their own processes).
2. If the tracee process is being already traced.
3. If the target process belongs to a set of process that cannot be traced (init and itself).

If all previous controls are satisfied, the monitor process becomes the parent of the traced process and the trace flag is set on its process descriptor. Then, as the previous case, the tracee is put in a non-interruptible sleep state by a SIGSTOP signal sent automatically by the kernel. This signal in fact kicks off the tracing mechanism, but it has been seen as possibile source of security issues as a process should not be aware of being traced [19]. In order to overcome this shortcoming the PTRACE_SEIZE and PTRACE_INTERUPT requests have been introduced in the 3.4 kernel version. PTRACE_SEIZE attaches the process specified in the pid to the process that has issued the request. Unlike PTRACE_ATTACH , PTRACE_SEIZE does not stop the process. To stop a process traced using it, an PTRACE_INTERRUPT request need to be issued. This cause the stop of the process without sending a SIGSTOP signal, making all the tracing process transparent to the tracee. The possibility to attach a running process is particularly useful when a process, which has been created by a different process from the tracer, has to be monitored. A typical example of this situation is when a tracee spawns a new thread. The newly thread must be attached using one of the previous requests as the trace flag is not inherited.

It is worth noticing that in both cases the tracer becomes the parent of the tracee process. The *task_struct* of a Linux process has two fields:

```
struct task_struct *real_parent;  /* real parent process */
struct task_struct *parent;       /* recipient of SIGCHLD, wait4() reports
   */
```

Listing 2.2: Parent and real parent fields with task_struct Linux

Usually both field point to the process which has created the current process. When a process is being traced the parent field is modified in order to point to the tracer process which becomes the parent of the child process for most purposes (e.g.,it will receive notification of child events and appears in ps(1) output as the child's parent), while the real parent remains the original one as invoking getpid() the pid of the original process is retrieved. In Linux a process has exactly one process father. Consequentially, this limits ptrace to track only one process per time making it a no-efficient mechanism for tracing multithread applications.

When the tracing mechanism has been correctly started, all system calls made by the tracee are intercepted by the tracer. The tracee process is put in a stopped state by the kernel each time it receives a signal or attempts to invoke a system call. Once the tracee has been stopped, the tracer is notified via a signal SIGCHILD and then it is allowed to access the tracee address space. The tracer may handle this signal either through a wait-family system call or by installing a dedicated handler. In the case of system call interceptor, the only relevant notifications are those regarding the attempt of invoking a system call. Precisely, the tracer should be notified twice, at the entry point before the system call is executed and at the exit point before the traced process is resumed. The sequence of events triggered when the traced process invokes a system call is depicted in Figure 2.1.

The cause of the stop in the tracee can be identified through the status variable returned by the wait primitive or the integer argument in the handler function. The Linux signal library provides different macros to easily deal with this signal status variable [ref signal]. The cause that determines the stop of the tracee process can be retrieved as follows:

```
WSTOPSIG(status) & (WITSTOPPED(status) == SIGTRAP)
```

Listing 2.3: Condition that identifies SIGTRAP signals

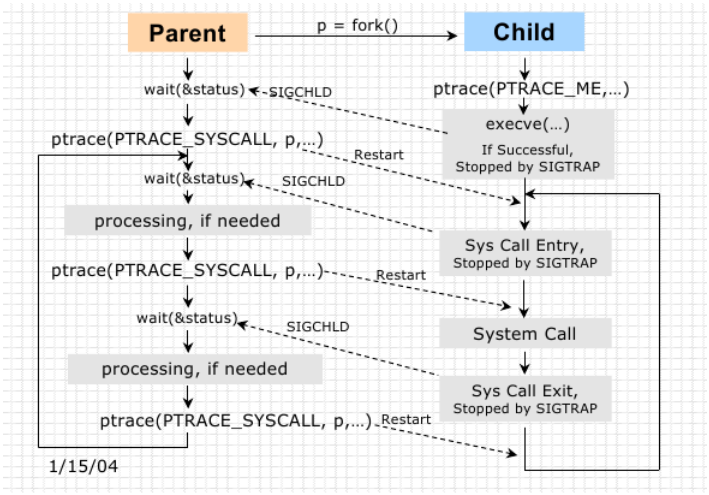

Figure 2.1: System Call invocation

Unfortunately, the status variable will assume the same value also when a SIGTRAP signal is delivered to the tracee for a different reason from invoking a system call.
This ambiguity may be solved in two different ways. Further information about the source of the signal might be retrieved by issuing a ptrace request with `PTRACE_GETSIGINFO`. Although this will introduce a further system call for each SIGTRAP signal delivered to the trace, which in the case of a traced process are the most common. This performance decrease can be avoided using ptrace with the `PTRACE_O_TRACESYSGOOD` option. This option ensures that when a signal trap is generated because the tracee is trying to make a system call the 7th bit of signal status

```
WSTOPSIG(status) & (WITSTOPPED(status)== SIGTRAP | 0x80)
```

Listing 2.4: Condition that identifies exclusively system call entry and exit

Now, the tracer can skip all false notifications and analyse only those regarding a system call.
One of the most important feature of system call interceptor is the possibility to identify the system call invoked by a process and retrieve all its arguments. The convention adopted in Linux **??** for calling a system call is to insert the number of the system call into the EAX register for x32 architecture and in the RAX register in x64, along with its parameters. The maximum number of system call parameters is 6 so the stack is not involved in passing arguments. The table below resumes the role of some registers during a system call invocation for both x32 and x64 architecture, these entire structures can be found in *sys/user.h*.

```
/* x64 architecture */
struct user_regs_struct
{
  ....
unsigned long int rdi; /* 1th parameter
    */
unsigned long int rsi; /* 2th parameter
    */
unsigned long int rdx; /* 3th parameter
    */
unsigned long int rcx; /* 4th parameter
    */
unsigned long int r9;  /* 5th parameter
    */
unsigned long int r8;  /* 6th parameter
    */
unsigned long int rax;
/* System Call number */
unsigned long int orig_rax;
unsigned long int eflags;
 ....
};
```

Listing 2.5: Linux structures representing the general purpose registers of x64 CPU

```
/* x32 architecture */
 struct user_regs_struct
{
  .....
  long int ebx;  /* 1th parameter */
  long int ecx;  /* 2th parameter */
```

```
long int edx;  /* 3th parameter */
long int esi;  /* 4th parameter */
long int edi;  /* 5th parameter */
long int ebp;  /* 6th parameter */
long int eax;
/* System Call number */
long int orig_eax;
  long int eflags;
  ....
};
```

Listing 2.6: Linux structure representing the general purpose registers of x32 CPU

While the tracee process is in stopped, the tracer can retrieve and modify the values of the tracee processor register via PTRACE_GETREGS/PTRACE_SETREGS. This methods offers only a way to access to the direct parameters that are conainet in the registers but offers only the possibility to retrieve the indirect parameters whose address is conatained in the resister. In order to retrieve the value of this parameter the tracee memory needs to be accessed. ptrace supports a a request to write into the tracee area adn one request to read from.

The tracee memory is accessed, before the system call is executed, to retrieve the system call identifier and arguments. This is used in software such as sandbox, for example [systrace, jail, c++] to check whether the system call invocation satisfies the requirements specified in a policy file or for auditing purposes in [systrace]. Furthermore, there is also the possibility to access the tracee memory when the process is stopped after the execution of the system call. This gives a means of analysing the system call return value, which is important for applying post policy rules in a sandbox application [jailer]. Post policy rules are used for those system call where a sensitive value is known only after the call is executed such as accpet, recv. Accessing the memory always introduces an overhead which has to be taken in account when performance is an important aspect of the application. Different approaches for accessing the memory of the tracee process are discussed in detail in 2.3.

Communications between the controller and target take place using repeated calls of ptrace, passing a small fixed-size block of memory between the two (necessitating two context switches per call); this is acutely inefficient when accessing large amounts of the target's memory, as this can only be done in word sized blocks (with a ptrace call for each word).[5]

retrieve the system call number and the direct arguments [2]. While in order to access to indirect arguments, the space memory of the tracee process need to be accessed. Ptrace offers different type of requests [27] to accomplish this, however all of them are limited in bandwidth to 4 bytes in x32 architecture and 8 bytes in x64.

Ptrace provides for the tracer the ability to fully control the execution of the tracee. The execution of the tracee can be resumed via one of the following ptrace requests:

PTRACE_SYSCALL : continues execution until next entry or exit from a system call.

PTRACE_CONTINUE : continues execution until the tracee receives a signal.

PTRACE_SINGLESTEP : continues execution until the next tracee instruction.

As we are only interested in intercepting a system-calls the execution of a tracee is resumed by calling ptrace with `SYS_SYSCALL` as argument, its execution continuing until the next system call event.

2.2 System call virtualization

Virtualization has become increasingly popular and a new demand for Linux is to act as a hypervisor. Different virtualization technologies have been developed to cope this increasing demand, such as instruction emulators QEMU, full or partial hardware emulation. System call virtualization is a technique to emulate the execution of a system call made by a process. Ptrace can be used to set up a system call virtualization mechanism in Linux. This approach has been followed by Jeff Dikk [two papers] developing UML whose goals was to run Linux kernel in user space for easing the debugging of kernel. A system call virtualisation needs a way to nullify system calls so that they would execute in such way as to cause no effects on the host. However, ptrace did not provide a means of aborting

[2] direct argument

a system call [annulling a system call]. This shortcoming has been fixed introducing the possibility to change to change the actual system call. Annulling a system call can be done by substituting the actual system call with a getpid, which introduces a small overhead due to its execution.
When a system call is virtualised, there is no need to intercept the exit point because the system call has been nullified. This prevents the execution of two context switches between kernel space and user space which yields to a 50% improvement in the performance. All these ptrace enhancements have been introduced in the kernel mainline and they can be used by calling ptrace with the parameter SYSEMU [ptrace documentation]. As it has already been said, this prevents the system call to be executed and notifies the tracing parent only at system call entry.
The tracing mechanism can be installed in a similar manner has described in the previous chapter. The only difference is that when the traced software is resumed issuing ptrace with PTRACE_SYSEMU parameter instead of PTRACE_SYSCALL.
System call virtualization is implemented by the tracing thread intercepting and redirecting process system calls to the system call handler. It reads out the system call and its arguments, then annuls this in the host kernel and executes the virtual system call instead. When the emulated system call is finished; the return values are stored in the memory of the traced process. This process is depicted in the figure.
An example of this approach can be found in [GOANA], where ptrace has been used to create a user-level file system development environment. The bulk of the application consists of a monitor called GOANA which intercepts all system calls made by the traced process. Once the system call has been intercepted this is substituted with a prototype function. This provides an easy way to test a file system prototype because we do not need to deal with massive body of kernel-level code and due to the fact it runs on user space the system can be analysed using a powerful debugger such as gdb which would have been impossible to use in kernel mode.
The performance is still the main problem of this approach as well. Even though by using SYSEMU we reduced the context switch between User Space and Kernel space to 2. There is still the necessity to access the memory at the system call entry point to retrieve its arguments and after the execution of the system call is finished to store the return values. This is a critical part of each monitor process, this problem as in the previous case will be treat in detail in the next section.

2.3 Memory access

The tracer process is executed in user mode, it thus is not allowed to write or read the memory space of the tracee process. The tracer needs to read the memory of the tracee process in order to retrieve the contents of the indirect parameters such as strings or pointers. There are two kinds of arguments direct and indirect, the former is contained in the general purpose register, while the latter is contained in the process memory and its address can be found into registers. The maximum argument number for a system call is six, so they fit in the register and there is no need to access the process stack.Also, the monitor needs to write the target memory address, for instance, when the system call is emulated and the outcome of the computation need to be stored in the traced memory space.

Ptrace

One method to access the memory of the tracee process is provided by ptrace. The monitor process may read from the monitor process calling ptrace with PTRACE_PEEKDATA when the target process is suspended. The writing procedure can be performed in a similar manner by using PTRACE_POKEDATA. Unfortunately, ptrace transfers only 4 bytes per time, therefore it has to be called many times to read or write a large amount of memory. Every call to ptrace requires a context switch from the monitor to the kernel and back, which decreases its performance since to make it, in fact, not a feasible way with a large amount of data.In order to improve the performance in accessing the traced memory space different approach can be taken.

Shared Memory

The monitor process has to ensure that the target process can access to the shared memory. In [Jailer]

the shared memory is set up using a preload library technique. The preload library uses mmap() to read-only map the memory region in the targets process space , in addition, it loads some code routine. Another technique presented in [Orchestra] consists of replacing the first system call made by the target process with an appropriate one such as shmget, ipc after making a backup of the values contained in the registers. This makes the target to run the new syst em call and attaching the shared memory to the target process. After performing this operation, the original system call is restored by using the register values previously saved. In both cases a small code is injected to the space memory which copies the content of a buffer to another one. When the monitor process needs to access to an indirect arguments, it can retrieve the base address from the registers by using ptrace and then use this routine to copy all the buffer in the share memory. In the Jailer system, the values of registers are modified to point to this memory area for security reasons [See race condition].

FIFO

Another approach proposed in [Orchestra] is to use FIFO structures. FIFO is inter-process communication which allows the monitor process to communicate to the target process by writing to and reading from the FIFO. The pipe can be created using the system call mkfifo()[15] and then it can be managed via I/O system calls usually used to deal with files (in fact the FIFO is a file in the file system). The target process can easily open the FIFO by using the FIFOs name as parameters calling open. While, if the target process has been spawned by the monitor process, it inheritances the file descriptor corresponding to the FIFO open by monitor process. However, there is still need to use a routine to copy the values from the target process space to the pipe. Other ipc mechanisms such message queue [jailer] can be used in a similar fashion.

Using /proc interface

The proc file system is a pseudo-file system which is used as an interface to kernel data structures. This interface may be used to collect information regarding the execution of a program such as file descriptors, memory layout and so on [ref]. Two files within this hierarchical file system are particular relevant for the purpose of accessing the memory of a traced processed. The first is `/proc/pid/maps` which contains a list of all the memory regions and their access permissions currently mapped in the address space of the process identified by the pid within the path. This feature is used to retrieve the location of shared libraries or shared segment within the address space of a process. It also used to identify whether the process is using a xVSDO mechanism to boost up the process of invoking a system call. The other file of interest is `/proc/pid/mem`, it has been introduced to provide an alternative to the ptrace approach discussed before. This file can be used to access the pages of a process's memory via the classic system call [3] used to deal with files. The two code fragments presents a possible way to implement this approach. Some constrains must be respected by the process that wants to access the memory of another process using this method.

- The process that wants to read from or write to `/proc/pid/mem` must trace the execution of the process identified by the pid via ptrace.
- The tracee must be in a stopped state.

If the previous checks are not respected an `ESRCH` error is retrieved. A cautious approach should be taken using this method as the ability of writing to the process's memory might be disable in some kernel versions. This is due to the confusion caused by the introduction of a vulnerability when the writing ability was first activated. The writing function within the kernel carried out a poor permission check which could be exploited by a user to gain root privilege [11]. However, a recent kernel (after 3.0) should be bug-free and support the writing function.

Cross memory attach

The space memory of a running process can be also accessed using *cross memory attach*, a new mechanism that has been recently introduced in the Linux kernel **??**. This mechanism is implement by two new system calls [25] that transfer data between the address space of the calling process

[3] The classic system calls used to handle files in Linux are open, lseek, read, write, close

and the process identified by pid. The data moves directly between the address spaces of the two processes, without passing through kernel space improving the performance respect to the previous approach. t has been successfully employed in [32] to access the system call's parameters.

For obvious security reasons, there are some restrictions that a process must respect in order to access the memory of another process. In order to perform the copy both process must have the same ownership or the calling process must have the `CAP_SYS_PTRACE` capability. The permission required is exactly the same as that required to perform a ptrace attach to another process.

2.4 Multi thread applications

In this chapter, the term multithread process/application refers to an application/process whose threads have been created using the system call *clone* [24] with the `CLONE_THREAD` flag.

Nowadays, most the applications are multithread. For example, a web server usually creates a new thread for each incoming request and let this newly created thread to accomplish it. This family of applications must be taken into account as possible target process for a system call interceptor and this requires a thoroughly analysis. As we already said in the introduction part, one of the most import requirements of a system call interceptor is to provide a means of intercepting all processes spawned by the traced process. Ptrace misses this aspect because it provides a tracing mechanism for single thread [ptrace documentation]; where each thread must be attached individually to the tracer process issuing a request with PTRACE_ATTACH. Moreover, a child process does not inherit the trace flag from the parent, so it can run untraced until the father attaches to it. These flaws make tracking multithreaded application difficult and error prone [race condition paper], so that some applications [jailer] decided to not support this kind of applications.

Nevertheless, ptrace offers a solution, though it is quite elementary. Setting the option PTRACE_O_TRACECLO
[insert a note for other arguments], the tracee will be stopped at the next clone() function and the tracer will start tracing the newly cloned process. This solution, in fact, is not a feasible way to trace multithread application when performance are important because each time a signal is delivered to a traced thread, all threads in the traced poll enter in a stop state.
This introduces an unnecessary latency because the other threads (those that have not received any signal) must wait until the event is processed by the tracer and the entire application is resumed before being able to carry on their tasks. However, in application where performance is not important, for instance debuggers, this represents a valid solution. In fact, this approach is used by GDB. Another issue related with this approach is that ptrace does not always guarantee that the forked processes are always automatically traced. Linux allows setting a flag on clone() (CLONE_UNTRACED) , which determines whether the child process can be traced or not. However, this can be easily solved noticing that the clone function will be intercepted by the monitor program and it can modify its arguments changing that flag to CLONE_PTRACE which makes the process traceable.
The common approach to deal with multithread application is to create a new tracer process [jain and sekar, jailer, orchestra, systrace,strace] for each new tracee process. Unfortunately, this solution raises a race condition which might yield to some system calls to escape from being intercepted. The new monitor is not allowed to trace the newly thread until the parent monitor detaches from it. When the parent monitors detaches from it the kernel sends a message to the tracee process and allows it to continue its execution. This leaves a small temporal window within some system calls might escape. This is critical problem in application such as sandbox, where all system calls must be intercepted for obvious security reason. A completely different approach is taken by strace where this problem is not tackled because a missed system will not comprise the entire application, while solving it will complicate the code, and the temporal window is rather small and the probability of missing a system call or more should be quite low.

A possible kernel enhancement, which solves this, would be to notify to the tracer when the newly thread has been created but it is not yet running. This would give the opportunity to safely attach a new tracer to the newly created thread without losing any call. Instead the tracer receives a tracing event only on the childs subsequent system call.

This problem has been addressed successfully in [Orchestra] as follows. When the tracee process spawns a new thread, automatically it is stopped by the kernel and the monitor start monitoring the newly created thread. At its first system call invocation, the monitor makes a backup of all new tracee process registers and then it substitutes the requested system call with a pause system call and finally lets it continue its execution. When the new process is resumed, it will enter in a pause state due to the injected system call. While this process is in the sleep state, the monitor is allowed to detach from it and the new monitor can safely attach to it. Once the new monitor has successfully attached, it restores the previous system call so that the execution of the new tracee process can continue. Although this solution works well, it introduces a small time overhead due to the execution of the pause system. Moreover, a communication between the new monitor and its father is needed to retrieve the backup information regarding the execution context of the new thread in order to restore the previous system call.

2.5 Aborting a system call

Aborting a system call

Having the possibility to abort a system call is an important feature. Lets consider the following scenario. A large application such as database has been sandboxed for security reason. It is possible that due to the nature of the application and the high number of system call requests, some policy checks fail. When this happen the system call should be nullified, but if this feature is not provided the only solution is to terminate the application even if there is not any real threat. Unfortunately Linux doesn't provide any means of aborting a system call. This lack has been one of the main reason because ptrace was not considerate a feasible way to implement a system call interceptor [Janus thesis]. Different solutions have been propose [C++ sandbox] and [Goanna], but this problem has been solved by [Uml Guy] who implemented a patch for the Linux kernel. This patch allows the tracer to substitute the invoked system call with another one by inserting a different system call id in the EAX register. Leveraging on the enhancement introduced for UML, the tracees system call can be substituted with the low-overhead system call getpid.
This technique for aborting a system call raises another problem, which value should be returned? One possibility is EINTR which represent the error code for system call interrupted. However, this is not a good choice because some application may be coded in such a way that, if they receive EINTR error they will retry the system call. If this happens in a loop the application gets stuck and the only way to break that loop is to kill the entire application. A better error code is EPERM, which stands for operation not permitted.

Chapter 3

Kernel-based tracing mechanisms

Kernel tracing mechanisms were originally developed in response to the limitations of the existing tracing mechanisms such as ptrace. Since the early 2000, several sandbox tools have been developed using a system call interceptor implemented within the kernel [43, 49, 34, 15] either using a kernel enhancement (ptrace++) or a kernel module (mod_janus).

Two different approaches can be taken to develop a system call interceptor in the Linux kernel. As the source of the kernel is available, a completely new tracing infrastructure might be introduced in the kernel exposing its tracing functionalities to user space as in the case of *ptrace* or at kernel space as in the case of*utrace*. This approach requires a massive change in the kernel source that is difficult to implement and error-prone due to the complexity of the kernel. Furthermore, it is not a portable solution as it will not be inserted into the Linux source (see case utrace),and thus it should be dispatched as kernel patch which is not easy to install and require to compile the entire kernel.

An easier approach is to insert the tracing mechanism within a kernel module. It simplifies the installation process because it needs only to be compiled and loaded in memory. It also has the benefit that the tracing mechanism can be activated when the module is loaded and deactivated when it is removed. In fact, this is the implementation choice widely used for building up kernel-based interceptors [43, 49, 34].

Inserting code in the kernel space is always a risky operation because there is the possibility to introduce some bugs which may compromise the whole system. In order to reduce this eventuality, an **hybrid interposition architecture** has been firstly developed in [43]. Then it has become the prominent approach to build kernel-based interceptor. The attribute hybrid is due to the fact that it is not a fully kernel based implementation, as it is composed of two components : a kernel-level enhancement and a user-level process monitor. The kernel-level enforcement is the core of the system call interceptor as its main task is to provide to the monitor process a means of controlling the system calls made by another program.

This usually was accomplished by overwriting the address of a system calls within the system call table[1] with the address of the respective instrumented system calls.An instrumented system call runs a pre-call and post-call routine as well as calling the old system call. The extra routines executed are usually referred to as **extension code** and allow tracing component to have a full control of the system call. It can prevent a system call to be execute, access its parameters or return values.

These methods rely on the fact the system call table is modifiable. Since kernel 2.6, this possibility has been removed, the address of the system call table is not accessible and the pages on which it is contained are now read-only. However, different tricks[2] to overwrite the system call table exist, but they are not a reliable solution for implementing a system call interceptor.

A solution for this problem may be to use the instrumentation features provided by the kernel instead of modifying the system call table. Instrumentation features are used mainly for debugging and tracing purposes but they can be effectively used for implementing a system call interceptor as well. A Kernel instrumentation tool is a mechanism that allows for a extra code, usually called as *extension code*, to be insert into a specific location in the kernel code. When this point is hit, the

[1]Kernel structure *sys_call_table* which contains addresses of system call handlers.

[2]For example. the address of the system call table can be retrieved from the file /boot/System.map

extension code associated is executed.

There are two type of the kernel instrumentation:

Dynamic the instrumentation code is inserted in the specific location at run time. Kprobes [41]

Static the point and the instrumentation code is declared in the source file. Kernel makers/trace point used in LTTng [7] .

The primary advantages of a kernel based approach is low overhead due to the intercepting mechanism. The overhead for a system call interposition is determined completely by the extension code executed. Moreover, the kernel code runs at the highest privilege level and as such, can access all kernel structure as well as the address space of a user-space process avoiding the overhead due to the context switch. However, the power afforded by a kernel-mode intercepting mechanism entails some drawbacks. The primary one is that the extension code, as it runs at kernel-level, must respect the constrains of this environment. In this environment some assumption commonly used in user-space may not be true in kernel space [such us such as dynamic memory allocation].This make a kernel developing complicated and error-prone. Introducing an error in the kernel has serious consequences, because it may compromise the entire system or introduce security flaws. In addition a system call interceptor based on the kernel requires super user privilege to be installed that may make it difficult to be used in a multi-user environment.

Other factor that make this approach cumbersome for implementing a system call interceptor is that the code is not easily portable among different architecture or kernel version. The process state represented by the status of its register is architecture dependent, the ABI used to pass arguments across different kernel functions may depend on the Kernel version. Therefore a system call interceptor realized using a kernel mechanism needs to be adapted for the architecture of interest.

In the remainder of this chapter the models previously introduced are explained in better details. The first model presented is *Utrace*, it is a kernel tracing architecture initially implemented as kernel enhancement to replace ptrace. Although, this has never been inserted in the main line of Linux kernel, it is an interesting prototype which is worth analysing [the interesting thing is the performance]. The third and the forth chapter introduce two different model of hybrid interposition architecture. The first is implemented using a filtering architecture the second using a delegating one. The last part introduces the use of kernel instrumentation mechanism, it is mainly focus on the Kprobe architecture explaining how this mechanism can be used to instrument system call in order to build up an efficient kernel-based system call interceptor.

3.1 Utrace

Utrace has been developed principally in order to overcome the ptraces limitations, especially those regarding performance and race conditions. Utrace is an in-kernel API which can be used to build kernel-based tracing mechanisms. It has been used to implement a secure sandbox for web application in [35], or as base for as virtualization mechanism in KMview [6].

The first difference with ptrace is that Utrace does not interact at all with user space, its interface its available only in the kernel space and all the extension code runs at kernel level. This implementation choice has been taken in order to avoid the overhead due to the context switch between user space and kernel space which is the major cause of low performance in ptrace.

The actors in an Utrace tracing mechanism are threads and tracing engines. A tracing engine is utrace's basic control unit, it is a piece of code defining the actions to be taken as consequences of an event occurring in the traced thread, such as invoking a system call. Typically, the utrace client is defined within a kernel module, and it establishes an engine for each thread of interest.

The Utrace interface provides the following basic facilities to build up an efficient tracing mechanism:

Event reporting: Utrace clients can register callbacks to be run when the traced thread issues a specific event of interest, i.e. system call entry/exit, exit, clone,etc.

Thread Control: The utrace client has full control of the running thread. It can inject signal, prevent a thread from running and abort a system call.

Thread machine state access: While the client is in a callback function, it can investigate the internal thread 's state by reading/writing the threads CPU registers and its memory space.

Utrace operates by inserting tracepoints at strategic point in the kernel code (these are called SAFE point more info) . When one of this points is hit by the traced kernel the callback associated with that event takes place. This callback, though happening in the context of the user process, occurs when this process is executing in the kernel space.

3.1.1 Setting up a System call interceptor mechanism using Utrace

The utrace client must be implemented in a kernel module in order to be able to access to the utrace interface. So, the mechanism will be activated when the module is loaded and deactivated when the module is removed. The tracing mechanism must ensure that a callback function is executed when the entry/exit system call event is triggered during the execution of the traced process.

A Utrace mechanism starts out by attaching an engine to a tread.

```
struct utrace_attached_engine * utrace_attach_task (struct task_struct * target,
                          int flags,
                          const struct utrace_engine_ops *ops,
                          void * data);
```

Listing 3.1: Synopsis utrace_attached_engine

Calling one of these function with the flag UTRACE_ATTACH_CREATE the engine is attached to the thread identified using its task_struck or PID. The structure utrace_engine_ops defines the callbacks function. In the case of a system call interceptor, the callback function of interest are those regarding the entry/exit of a system call.

Once the engine has been attached, the SYSENTRY and SYSEXIT need to be set in the engine. This can be accomplished using the following function :

```
int utrace_set_events (    struct task_struct *,
                           struct utrace_engine *,
                           unsigned long eventmask);
```

Listing 3.2: Synopsis utrace_set_events_task

Once the setting phase is completes, each time the traced process attempts to invoke a system the callbacks will be executed. During the execution of the callback function the tread is put in a QUIESCENCE status. This means that is stopped and will not start running when its status is accessed. Each callback takes as argument the task_struct which represents the state of traced process before the event has been triggered. Retrieving information from this structure such as system call number arguments depends on the architecture on the CPU architecture (i.e. x86 and x86_64 have different registers). The thread then can be resumed or aborted depending in the value returned by this function.

One of the important characteristic of a system call is to retrieve both direct and indirect arguments of a system call. The direct arguments can be retrieved from the CPU register. While the indirect ones are in the address space of the traced process and only their address can be retrieved from the CPU registers. The kernel provides two routine which allow to read (copy_from_user) from and write (copy_to_user) to the address space of a user process. Using these routines a value can be copied in the kernel space and analysed.

The system call interceptor, described so far it does not handle the case when the traced process spawns a new process. Utrace provide an event and its associated callback to handle this situation. If the traced process attempts to invoke a fork/clone system call, the CLONE event callback is called when the new child thread has been created but not yet started running (Note that this is the solution proposed in the previous chapter for ptrace in case of multi thread application).The newly thread

cannot be scheduled until the CLONE tracing callback return. This allows the tracing mechanism to create a new tracing engine and attach it to the newly process, ensuring that all system calls are correctly intercepted and analysed.

The main advantages of utrace is its performance. [Nice some examples]. Even though utrace seems a good solution for implementing a system call interceptor, it has been harshly criticized due to its kernel-based model. It suffers, as all kernel-based mechanisms, of the no-portability problem. It was intended to be a the ptrace killer application, but this is not happened because its interface can be used only within the kernel. A user need to have a base knowledge about kernel programming to write even an easy interceptor, which is not a common skill. These arguments caused the abandonment of utrace's development and it has never joined in the kernel main line.

3.2 Kernel hybrid interposition architecture

The first kernel hybrid interposition mechanism has been implemented in the second version of Janus sandbox [43] to provide a means of monitoring and modifying the system calls made by the sandboxed process. The necessity to develop a new interceptor mechanism raised from the limitation of ptrace in effectively aborting a system call. Before the enhancement introduced to support UML [9, 8], the only way offered by ptrace to prevent a system call from being executed was to terminate the entire program. This, obviously, was not a feasible solution for a sandbox tool because some false-threats can be thrown even though there is not any real threat. Event though the sandbox implemented with this intercepting mechanism is prone to vulnerabilities[ref], the same interceptor mechanism has been reproduced in other sandoxes tools such as MapBox [2], Systrace [34] with some improvements.

An hybrid system call interceptor is composed of a tracing engine which completely resides within the kernel whose task is to track all system calls made of the traced program. It allows a monitor process to access the tracing features through an easy interface in user space .Typically the communication between the monitor process and the tracing engine takes place via a char device, for example in mod_janus is /dev/fcap. Once the tracing mechanism has been correctly set up all trap events associated with the traced can be retrieved by the monitor process through a select or poll system call. As in the case of ptrace, a relevant overhead is introduced due to the context switch between user space and kernel space. However, in the case of a kernel interceptor mechanism, this shortcoming can be mitigated by leveraging on power and flexibility of this method . For example, it allows a fine-grained control over the system calls allowing a monitor process to intercept only certain calls while leaving the other unmonitored decreasing the overall overhead.

This may look not a good choice for auditing or record-replay purpose because all system call and their parameters need to be recorded. That is true, but another important characteristic of a kernel interceptor mechanism is its flexibility. If we want to use this kind of interceptor for record-replay purpose, the monitor mechanism can be inserted within the kernel space reducing the overhead just to the execution of the extension code.

As describe in the previous few line the kernel approach is extremely powerful and flexible, though placing an entire system call interceptor in a kernel is not a trivial process and it can introduces errors and new vulnerabilities that can compromise the entire system.

In the remainder of this section we analyses the system call interceptor implemented in in the second version of Janus sandbox. This has been chosen because other system call interceptors are implemented in a similar fashion and its source code is available on line.

The entire system call interceptor is implemented within a kernel module called mod_janus. To be able to use it the module must be loaded inside the kernel memory. One the module is correctly loaded in memory, the system call table is saved and a char device is created in the dev directory. This char device is used to carry out the communication between the intercepting engine within the kernel and the the monitor process in user space.

Before starting tracing a process, the monitoring process need to allocate the resource for supporting the tracing operations. This is accomplished by calling the open system call on the char device /dev/fcap, which returns a descriptor representing a monitor structure that can be used to track a

process. Once the monitor has been created, the monitor process is attached to the traced process by issuing a request via ioctl with parameter BIND and the descriptor of the monitor previously created. Furthermore, this function takes two additional parameters that give the possibility to the monitor process to specify for which system call entry/exit it will be notified. This is the first difference and improvement respect to the tracing mechanism provided by ptrace, which is based on an approach of all-or-nothing.

The system call interceptor is installed by overwriting the system call's addresses within the system call table with the address of a redirecting routine. The main task of this routine is to perform some preliminary checks (such as whether the process is already traced) and redirect the program flow to the tracing engine. This must be really concise and fast as it is invoked even by programs that are not being traced. It is usually implemented with few lines of assembler code.An interesting example of such routine can be found in the Janus source [ref source] in the file ent.S.

When a program issue a system call that has been replaced with the routine, the system call is not executed and the control flow is redirected to the tracing engine. The first check taking place here is whether there is a a monitor associate with this process. If the result is positive a request for a EVENT_CALL_ENTER is issued, otherwise the real system call is executed. Let's consider the case of the system call has been called by a traced program.When the event EVENT_CALL_ENTER is issued the traced program is put in a deep sleep as well as notify this event to the monitor process to its next call to either select or pool system call. The event type then can be retrieved with a read system call on the file description. While being in a sleep state the traced process can be accessed by the monitor process, as in the case of ptrace there are several different ways to access the memory of the tracee process, those one presented in the section 2.3 are still valid also in this case. However, for completeness of the subject we reported the method adopted by Janus. When the process is stopped the monitor process can request access to the system call's arguments by issuing a request via ioctl with argument FC_IOCTL_FETCH_ARG. When the monitor calls this function, it must specify the argument to be retrieved, its type[3] and a user space buffer on which the argument will be stored. Particularly interesting is the type TYPE_PATH_FOLLOW, when this option is specified the path name returned is expanded and canonicalized by the kernel preventing race condition on the path name [**?**].

Once the monitor has terminated its operation on the tracee process its execution can be resumed or denied specifying the right argument on a write request on the descriptor. If the tracee process is allowed to continue the system call is executed. Moreover, if a exit trap for this system call has been specified a similar sequence of event as that one describe above takes place when the execution of the system call is completed. The only difference is that the event issued is the EVEN_CALL_EXIT.

An exception of the previous approach is the fork/clone system call. The exit point of this system call is always trapped, the pid of the newly created process is retrieved from the task structure of the father (field p_cptr) and then it is stopped before it can make any system call. This allows the monitor process to start monitoring this process as well.

(I am not sure about this the code looks like there is a temporal windows within the child process might invoke some system calls)

It is worth mentioning the concurrent strategy adopted in this intercepting model. The monitor process can handle multithread application using a multiplexing model, in which a single monitor listens to the events associated to all threads of interest at the same time. This model may be implemented by creating a set of file descriptors on which all thread descriptors are inserted. Then the select function may be used to listen to the pending requests on these file descriptors. This implementation choice has been made because it would significantly reduce overhead over load. However, as showed in [15], this reduces the scalability of the system as the single monitor becomes the performance bottle neck.

[3]type indicates the type of the argument being requested, for example TYPE_SCALAR will store a scalar argument in the destination buffer, while TYPE_STRING will store a string

3.3 Kernel delegating architecture

The interceptor model presented in 3.2 suffers of a series of security problems (race condition [ref]) which makes its design and implementation substantially error prone to race conditions. This is due to the fact the action of checking a resource such as the system call parameters and the action of use is not an atomic operation therefore the resource may change between the two moments. This is usually called TOCTOU race condition. In order to overcome the limit of this architecture an alternative intercepting architecture has been developed in [15]. It is usually referred as *delegating architecture*.

A delegating architecture is composed of tree main components.

Kernel module: A small kernel module whose task is to prevent system calls made by the traced program from being executed. Furthermore, it also provides a trampoline instruction that redirects the system call back to the emulation library. (This can be easily accomplished overwriting the system call table for example).

Emulation library: The emulation library resides in the program's address space. When a system call is made by the traced program, the trampoline instruction within the kernel module is hit and a call back to the specific handler in the emulation library is issued. Then, the emulation library converts this system call request into a IPC request to the agent. In addition, to boost up subsequent system calls from the same point in the program execution, the handler analyses the instructions, if they have the expected form, applies a runtime patch to jump directly to the handler. This avoids the expensive context switch from user space to kernel space and back for subsequent system calls.

This library needs to be installed in the program's address space before the program starts running,so that all its system calls are intercepted and executed through the agent. The solution adopted in [15] is to modify the ELF loader so that the program is loaded in memory only after that the emulation library has been installed.

The communication between emulation library and an agent take place over a UNIX domain socket. This communication model has been chosen because it allows a file descriptor to be passed between the process and the agent. This is a crucial feature as it allows delegation of accesses to resources (i.e. open files and socket), while permitting the process to use them directly.

User-level agent The user-level agent is responsible for handling request for system calls from the emulation library. This is the most [delicate] and complex component, it must executes system calls on behalf of the traced process as well as providing a normal system call interface, as the monitored process should not be aware to be tracked. However, the Linux system call interface is rather complex and accomplish this is not an easy task. In [15] the only system calls analysed are those allowed in an sandboxed environment, and this does not offer a comprehensive view of all issues which may raise using this system call interceptor. For example, the case where the traced process invokes a mmap system call is not analysed. This may raise an issue because the new memory is attached to the process who invoked the system call, which is the agent in a delegating sandbox. We try to cover all problems linked to the delegation agent by subdividing the system call in subgroup and providing an possible implementation which should not be bound to sandboxing environment.

System calls fall in few subcategories :

Process context dependent There are few system calls that they result is bound to the execution context, as the agent resides in a different execution context this rises an issues. For example, in the case of the client calls mmap this must be executed by the agent, but the new memory space is attached to memory of the caller that in this case is the agent. For this case, a possible solution is that the agent modifies the argument of mmap so that the new memory area is shared between the two processes. [there may be more problematic cases].

Resource access In Linux resources are accessed via descriptors. Applications starts with an file descriptor space containing only input,output and error file descriptors. To grant access to an additional resource the monitored process must execute a system call (i.e. open, socket), that then will be intercepted and executed by the agent. The resulting descriptor is passed to the monitored process via Unix domain socket. Once the resource has been correctly opened, the monitor process can modify the object referred to by the descriptor by passing it to the agent, for example this happens in the case of ioctl or bind system call.

Id management The process's identity is represented by the user and group id. To perform accesses on the process's behalf the agent must assume the identity of the monitored process. This is accomplished by reproducing the identity state of the client within the agent, and all system calls (setgid, getuid) update this internal state. When the agent performs a call on the client behalf's, it assumes the identity saved in this internal state.

Signals The monitored process's signal are send by delegating the call to the kill system call to the agent.

Spawn new process When the client process invokes a system call such as clone or fork, the emulation library notifies this to the agent. The agent then spawns a new agent process and returns a new domain socket to communicate with the newly agent.Finally, the monitor process via the emulation library calls into the kernel to execute new fork.

This model has been mainly developed to overcome the securities issues in the filtering model. The structure of the delegating model itself solves the problem linked to the TOCTOU as the system call are invoked from the agent which resides in a different address space from the monitored process. This make impossible for an the multithread process to change the argument of a system call after this has been delegated to the agent. [More info] [More info about the overhead due to the delegation]

3.4 Kernel Probes

Kprobe [28] is a simple lightweight instrumentation mechanism developed by IBM and it has been introduced in Linux Kernel Version 2.6.9. Kprobes allows a user to dynamically insert a *probepoint*[4] in a specific kernel location. When a probepoint is hit a user-defined handler is executed. This will be executed in the context of the process where the probepoit has been hit. Kprobes has been mainly used for debugging purposes because a debug routine can be inserted easily in the kernel without recompiling it, for kernel tracing application as SystemTap[1], for performance evaluation, for fault-injection,etc.

Kprobes operates by overwriting the first byte of the probed instruction with a breakpoint instruction (e.g., int3 on i386 and x86_64). The original instruction is copied into a separate region of memory.When the probed point is hit by the CPU a trap fault occurs, the CPU register are saved and the control passes to the Kprobes manager via the kernel notification chain[5]. Then, the Kprobes manager executes a user-defined routine *pre_handler*. After that, the original instruction must be executed. This is run in single-step mode out of the normal program flow. This solution called *"single-step out of line"* or *"execute out of line"* (XOL) allows a probe mechanism to work with multiple processes at the same time. [more info] When the execution of the probed instruction is completed, the control returns to the kprobes manager which executes a user-defiend post_handler. A nice description of the kernel probes can be found at [41].

A probe point can be registered using the function *register_kprobe()* specifying the address where the probe is to be inserted and what handlers is to be called when the probe point is hit. Recently,the possibility to insert probe point through symbolic name has been introduced in the kernel. This facility is particularly useful as a routine can be probed just using its name. Currently three different types of kprobes are supported :

[4]Probepoint is the address where the instrumentation is registered

[5]The kernel notification chain is the communication systems used within the Linux kernel, it follows a **Publish-subscribe** model.

Kprobe Kprobe can be inserted at any location within the kernel.

Jprobe Jprobe is inserted at entry point of a kernel function and it provides a convenient way to access the function's arguments.

Kretprobe Kretprobe , usually called return probe, is inserted at the end point of a kernel function.

The over head introduced using kprobes is to be taken into account when the performance is a important part of an application. The overhead is principally due to the execution of two exceptions for each probe instruction. A series of kprobes, called kprobe-booster, has been developed and integrated in the kernel to reduce this overhead. Their implementation rely on the fact that the post_handler is not always used and, if so, the second exception can be avoided using a jump instruction. This enhancement reduces by half the overhead due to kprobe instrumentation. This result was easily imaginable because the number of execpetion has been halved. A further improvement has been proposed in [33] where jump instructions are used instead of the break instruction. They claim to have achieved performance 5 times better than a normal probe.

3.4.1 System call interceptor using kernel probes

[to write]

3.5 Seccomb-bpf

[to write]

Chapter 4

Binary Rewriting

Binary rewriting is a software technique that transforms an executable by maintaining its original behaviour, while improving it in one or more aspects such as runtime performance, security and reliability. The research literature is full of examples of binary-rewriting methods applied to a variety of topics, including optimization [36], code instrumentation [22, 30], security enhancements [13], software caching [31, 4] and emulation [3]. Consequent to the advantages of binary rewriting, numerous binary rewriters have been developed. Typically they fall into two main categories *static rewriters* and *dynamic rewriters*, which can be further subdivided according the approach taken in the rewriting process, *full rewriting* where the entire binary is rewritten and *selective rewriting* where only relevant instructions are modified.

Static binary rewriters modify the executable file off-line allowing them to perform complex analysis and transformations. Due to their offline nature, *relocation information* is required to identify code regions and address locations. The identification of code regions is necessary to ensure a full disassembly of the binary file. Disassembling the entire code segment is an insufficient solution, as compilers often insert data within the code segment (such as jump tables and padding). The identification of address locations is required to adjust addresses during the relocation phase. The necessity of relocate information can be avoided using a selective binary-rewriting approach [30, 16] as the original code remains the same to the greatest extent possible. However, this approach allows only minimal transformations such as insertion of a trampoline instruction to jump into an instrumentation block and peephole optimizations. [1]

Dynamic binary rewriters modify the executable during its execution. The key advantage of this methods is that it does not require any relocation or symbolic information as at runtime the identification of the code regions and the address locations is straightforward. Unfortunately, dynamic rewriters introduce a significant overhead during execution, as all rewriting operations occur at runtime. This makes it infeasible to use a binary rewriter to perform complex transformation such as automatic parallelization as the overhead introduced is prohibitive. Consequent to this limitation, dynamic binary rewriters have been only employed for simply code instrumentation, and even in this case the overhead introduced is significant, for DinamoRIO 20% and PIN 54% as reported in [39].

Despite their limitations, both static rewriting and dynamic rewriting can be successfully used to intercept system calls. In this section we focus on ELF[2] binary files that run on the Linux platforms, including both x86_32 and x64 architectures. Several instrumentation tools have been implemented [22, 30, 45], but none of them has been exclusively designed for intercepting system calls, though they can be used for accomplishing such a task.

A system call interceptor based on binary rewriting, works by placing a branch instruction at each system call invocation within the application code to transfer control to the instrumentation

[1] A peephole optimization is an optimisation performed over a small set of instructions. It works by recognising sets of instructions that can be replaced by shorter or faster set of instructions.

[2] Executable and Linkable Format (ELF, formerly called Extensible Linking Format) is a common standard file format for executables, object code, shared libraries, and core dumps.

code. As in the case of previous interceptor mechanisms, a way to pre and post process a system call invocation needs to be provided. This is accomplished by the instrumentation code, which executes a pre-routine, the system call and a post-routine in sequence, and then returns control to the application. A routine must be designed in such way to ensure that the program state is preserved across its execution, for example saving the state registers in the stack before their execution and restoring their values afterwards. This guarantees that the behaviour of the modified binary is the same as the behaviour of the original file. In addition, this system call interceptor must be able to identify all instructions that issue a request for a system call. Linux provides three different ways to invoke a system call:

Interrupts have been used by Linux to implement system calls on all x86 platforms since its first release. To execute a system call, a user process can copy the desired system call number to the eax register (both x86_32 and x64) and execute the instruction `int 0x80`. This generates the interrupt 0x80 and the corresponding interrupt service routine is called. The task of this routine is to save the current state and call the appropriate system call handler based on the value of the eax register.Further information regarding the implementation of this routine can be found in the *entry.S* file in the source of the kernel. However, this mechanism has turned out to be really slow since Pentium IV architectures, arising the need of a new efficient way to make system call.

```
movl    $len,%edx
movl    $msg,%ecx
movl    $1,%ebx
movl    $4,%eax
int     $0x80
```

Listing 4.1: Write system call invocation via interrupt on x86_32 architecture

Sysenter/Sysexit instructions were introduced by Intel from the Pentium II and later CPUs to fix the overhead issue correlated to the use of interrupts as mechanism to invoke system calls. `SYSENTER` can be executed by any application, while `SYSRET` can only be executed by ring 0 programs.These instructions are used as a fast way to transfer control from user mode (ring 3) to privileged mode (ring 0) and back quickly, allowing a fast and safe way to execute system routines from user mode.These instructions directly rely on the *Model Specific Registers* (MSR) [**?**] which complicates that way to invoke a system call for an user. Instead of using this mechanism directly, it is strongly recommended to use the system call entry/exit point exported in user space by the kernel since version 2.6 as presented in Listing **??**. Kernel creates a single page in the memory and attaches it to all processes' address space when they are loaded into memory. This page contains the actual implementation of the system call entry/exit mechanism and it is called *linux-gate.so*. This mechanism is usualy referred to as *Virtual Dynamically linked Shared Objects* (VDSO). It was first introduced to reduce the calling overhead on simple kernel routines (i.e. `getpid()` and `gettimeofday()`). Then it has been extend to work as a way to select the best system call method available in the architecture. This approach is available only on x86_32 architecture as x64 architecture provides an efficient instruction that satisfies the system call invocation requirement.

```
movl    $len,%edx
movl    $msg,%ecx
movl    $1,%ebx
movl    $4,%eax
call    *%gs:0x10
```

Listing 4.2: Write system call invocation via VDSO gate on x86_32 architecture.Note that the offset may change in a different platform.

Syscall is a new instruction that has been introduced to support fast system call on x64 architectures. The number of the syscall has to be passed in register rax and its arguments go into the other general CPU registers, similarly to the interrupt approach. Returning from the syscall, register rax contains the result of the system-call. In addition, this new instruction fully supports six arguments, therefore no argument is passed directly on the stack. A short code sample that uses this method for making a system call is presented above.

```
mov    $1,   %rax
mov    $1,   %rdi
mov    $mes, %rsi
mov    $len, %rdx
syscall
```

Listing 4.3: Write system call invocation via syscall on x64 architecture.

A system call interceptor whose aim is to work with both x86_32 and x64 files must be able to correctly identify all previous instructions. When using a rewriting approach to intercept a system call, all libraries linked with the program must be instrumented as well to ensure that all system calls are caught. In the rest of this chapter we discus how to implement a system call interceptor using a binary rewriting approach, both dynamic and static. Section 4.1 introduces in more detail techniques for static binary rewriting as well as issues that arise when a static rewriting is used for modifying ELF binaries within an architecture which supports a set of instructions with different length (such as Intel x86 and x86_64 instructions). Section 4.2 presents the general approach adopted by most of the dynamic binary rewriters.

4.1 Static binary rewriting

Static binary rewriting is a powerful technique that transforms a binary file such as executables or libraries into a partially or totally new file. It allows to perform complex transformations (for example, automatic parallelization and porting binary between different architectures) as well as insert instrumentation code (for example, for tracking purposes and security enhancements).Binary instrumentation offers a way for monitoring and controlling the execution of an instruction by allowing a extra code to be executed before or after of the instruction of interest. In this section, we describe how static binary instrumentation methods can be employed for the purpose of intercepting system calls. Although static code instrumentation introduces a minor number of changes than other binary transformations, it must address all issues that affect a generic static binary rewriter. There are numerous challenges that a static binary rewriter must address in order to ensure effectiveness as well as correctness, the largest of which include how correctly interpret the instructions within the text segment, how to relocate the code maintaining the original behaviour, how to accommodate the extra code needed by the extension routines and how to deal with variable-length instructions.

The first step of the rewriting process is the identification of all instructions, this process is usually referred to as *code discovery*. Unfortunately, identifying all instructions cannot be accomplished merely by disassembling the text segment of a binary file, as compilers often produce a ELF binary whose text segment might contain data. Compilers usually store small data structures for providing convenient and efficient lookup of data such as identifiers and descriptors.In order to guarantee correctness of the executable, the portions of the text segment containing instructions must be identified as mishandling the data within the text segment might result in a different application's behaviour from the original one.

Diverse code discovery algorithms have been designed such as *linear sweep*,which disassemble each location in a linear fashion, but it does not guarantee 100% disassembly. PEBIL uses a *recursive traversal*, which identifies instructions by following only valid control flow edges. When control transfer instructions are encountered the algorithm continues discovery at the target location. A difficult problem is raised when an *indirect* transfer instruction (e.g. `jump _entry`) is identified as the

target address is not identifiable statically. Different techniques have been developed to overcome this problem:

- The easiest solution is to rely upon *relocation entries*[3], unfortunately these are not included in commercial applications which narrows the applicability of this approach to few application.
- A different approach that does not rely on relocation entries is *peephole examination*, used in [22]. Peephole examination consists of analysing the instructions nearby the indirect transfer function in order to determine the target address of the indirect branch. This methods works, but it cannot guarantee a 100% code discover.

Another problem to be addressed by a binary rewriter is that some instructions will contain operands which reference other location within the binary file. During the rewriting process the target of these instructions might change due to the relocation of the referenced block. Load and store instructions reference to locations containing data. Maintaining the original data segment and all data portions to their original address in the rewritten binary ensures that the these instructions can be relocated without changing their operand. While the operand of call and jump instructions contain the address of other instructions. If the address of the target instruction can identified statically, it can be adjusted to point to the relocated instruction. In the case this is not possible, the address of an indirect branch must be identified using the techniques discussed in the code discovery code phase.

A common approach to transfer control from the application code to the instrumentation routine is to replace a single instruction or more with an unconditional branch instruction that performs the transfer. On platform with fixed-instruction length instruction sets, such as MIPS, this approach is straightforward as all instructions can be replaced with another one without consequence. While on platform that uses a *variable-length* instructions (i.e, x86), it may not always be possible to instrument an arbitrary point using a replacement technique. The amount of space available at the instrumentation point may be insufficient to accommodate an unconditional branch instruction, large enough to reach the instrumentation code.

In x86 architecture both x32 and x64, an unconditional branch that uses an offset of 32 bits requires 5 bytes. The instructions of interest for a system call interceptor introduced in the previous chapter, are of size 1 for `INT` and 2 byte for `syscall/systenter`.Therefore, simply replacing a system-call instruction with a branch to the instrumentation code is not feasible way to accomplish the instrumentation of all system call invocations within the binary file. Different techniques have been used to transfer control to the instrumentation code.

The first alternative is the method proposed by the BIRD project [30]. When the instruction at the instrumentation point is shorter than 5 bytes, additional bytes could come from the first one or two instructions immediately following or preceding the instruction at the instrumentation point as long as doing so does not affect the programs execution semantics. In general, an instruction is safe to be replaced if it is not the target of any branch instruction. When BIRD cannot find any additional safe bytes to locate the branch instruction, it uses an interrupt instruction (i.e. `int3`). This is instruction fits perfectly the instrumentation requirements because it is of a single byte size and transfers the control to an arbitrary routine via the exception handling facilities provided by the operating system. Unfortunately, it is unsuitable for accomplishing this efficiently as the INT introduces a large overhead due to the heavyweight switch from user space to kernel space and back. A similar approach has been used in *seccomsandbox* in a dynamic fashion.

Another option is the method presented in PEBIL[22], which is ideal for implementing a system call interceptor as it allows to insert a instrumentation code at an arbitrary points within the code application. The instrumentation code might be inserted before and after a system call instruction in order to control its execution. It uses a relocation and reorganization of the code at the function level in such a way that guarantee that 5 bytes are always available at the instrumentation point. This process consists of 4 different steps :

[3]Relocation entries are generate by the compiler when an absolute address is not available during compiling time, for example a function in a library

1. *Function Displacement* relocates the contents of a function to an area of the text section for use of the instrumentation test. Then the original entry point of the function is linked to the new location via an unconditional branch.

2. *Branch conversion* The address of all branches are extended to use 32-bits offset in order to make them able to reach every target within the application memory space (typically 4Gb)

3. *Instruction Padding* pads each instrumentation point with enough empty space so that a 5-byte instruction can be inserted.

4. *Instrumentation* replace the instructions at each instrumentation point with a branch that transfer control to the instrumetation code.

This method can be specialised for the purpose of system call interceptor by relocating only the function which contains a system call invocation. The padding instructions should be inserted before and after of a system call invocation. This allows a pre-routine to be inserted before the system call invocation and one just after the system call. Note in this case the instrumentation code does not execute the system call, while in the previous solution the system call instruction was rewritten and thus it must be executed by the instrumentation code.

```
0000c000: <foo>:
c000: 48 89 7d f8 mov %rdi, -0x8(%rdb)
c004: 5e       pop %rsi
c005: 75 f8      jne 0xc004
c007: c9       leaveq
c008: c3       retq
```

Listing 4.4: Original instructions, x64 architecture

```
00008000: <_rel_foo>:
    8000: 90 90 90 90 90    [empty space]
    8005: 48 89 7d f8       mov %rdi, -0x8(%rdb)
    8009: 90 90 90 90 90    [empty space]
    800d: 5e                pop %rsi
    800e: 0f 85 f8 ff ff ff jne 0x8009
    8014: 90 90 90 90 90    [empty space]
    8019: c9                leaveq
    801a: c3                retq

0000c000: <foo>:
    c000: 48 89 7d f8       jmpq 0x8000
    c005: 90 90 90 90       [empty space]
```

Listing 4.5: Instructions after the rewriting process using a relocation code approach at functional level

A totally different approach is that taken by binary rewriters that performs a full rewrite of the entire file as in the case of *SecondWrite, REINS*. In this case the problem of finding new space for the instrumentation code does not occur as it can be inserted during the rewriting process. Usually, following this approach a binary file is translated in a intermediary language, optimised and then translated back to the binary format. For example, *SecondWriter* integrates binary rewriting technologies with the complier LLVM [**?**] by translating the binary file into LLVM's intermediate language IR. When the application is represented in IR, is passed through a series of analysis and transformation. The result then is given an code generation routine which recompile the application.

When a selective rewriting approach is used such as in PEBIL and BIRD, the problem to insert the instrumentation code within the binary file raises. In order to insert additional code and data into an executable, additional space needs to be allocated within the executable in such a way that

will be correctly treated at load time. Two additional segment should be inserted, one containing the instrumentation code and the other containing the data required by the instrumentation code. Furthermore, in the case of the PEBIL approach also the size of the code segment of the original binary should be modified as more space is needed to support the padding instructions. The ELF code segment might be extended using similar approach as for ELF injection [**?**]

The static binary approach for implementing a system call interceptor is more difficult to implement with respect to the previous approaches discussed in this document as require to effectively solve all previous shortcomings as well as ensuring the correctness of the translated binary. In spite of the implementation difficulties, it provides a very low overhead due to the tracing mechanism, around 5%. This intercepting mechanism does not require a monitor thread as the "monitoring code" resides in the same memory space of the controlled process. This is an remarkable advantage as it avoids all overhead due to accessing the memory of another process. Furthermore, it can be used also to implement a delegating architecture where a binary file is statically patched in order to issue requests for a system call to another process, as in the case of [16, 13] (they use a dynamic rewriting technique but the same approach can be implemented in a static manner).

In the case of this typology of system call interceptor is used for security purpose, it must be ensured that the instrumentation code cannot be rewritten by the application's instruction. This can be accomplished through *software fault isolation* techniques [44]. Only instrumenting the binary file is not sufficient to guarantee that all system calls are intercepted as, usually, programs use wrapper functions provided by an external library to make requests for external resource. The libraries linked with the program must be instrumented as well. This can be done by implementing a specialised version of elf-loader, that inserts the code in the libraries before loading them in the memory. An alternative could be to statically link the program to the libraries and then perform the binary instrumentation on the entire file which contain application and libraries.

4.2 Dynamic binary rewriting

Binary dynamic translation is a technique for dynamically modifying a program as it is being executed. It has been used in a variety of different areas: binary translation for executing programs on non-native CPUs [10]; fast machine simulation [47]; and recently, dynamic optimization and safe execution [4, 39, 13]. In this section we describe how software dynamic translation can be used to implement system call interception.

Most software dynamic translators can be seen as a virtual machine. A dynamic translator fetches instructions, performs an application-specific translation, and then arranges for the translated instructions to be executed; these are roughly the same actions performed by a virtual machine. Implementing a system call interceptor application in a software dynamic translator is a simple matter of overriding the system call instructions (e.g. syscall) with a jump to a system call handling routine that performs user defined routine. This routine can be implemented in a similar fashion as those within a static binary approach, although when using a dynamic rewriting the problem of extending the ELF binary does not raise. A routine can be loaded in memory before running the application, and the memory segment on which it is located can be protected through the security features provided by the operating system (e.g. `mprotect`).

In a binary dynamic rewriter, the applications code is translated on demand and thus the only translated code is the executed one, while in a static translator all the code must be already translated at run-time. This helps to handle easily indirect jumps as the operand of these instructions can be identified from the execution context. Dynamic binary rewriter is slower than one that use a static approach for two reasons.The first reason is that the translation is performed at run-time so the time spent translating the code is to be added to the time spent executing it.This implies that the translation must be performed as fast as possible. It also means that the benefit from executing translated code must overcome, by far if possible, the time spent translating code. The second reason derives from the first, as the time spent in the translation must be the less possible most of operations available for a static rewriter cannot be performed following a dynamic approach as they would introduce a prohibitive overhead. Although this limits the area of use of a dynamic rewriter, a system call

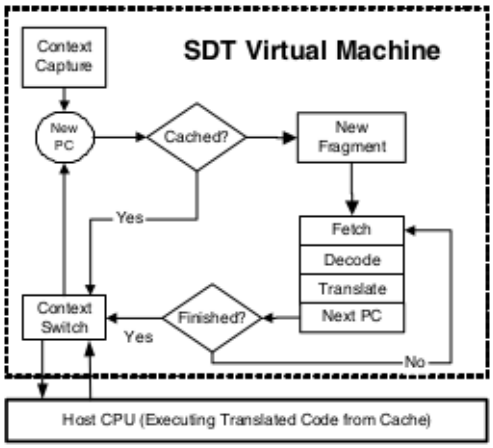

Figure 4.1: Dynamic translation algorithm

interceptor based on dynamic rewriter can be implemented.

A dynamic rewriter mediates the execution of an application by examining and translating its instructions. The basic unit of translation is a *fragment*. A fragment is a sequence of instruction ending with a control instruction. At the end of each basic block the applications machine state must be saved and the control returned to the dynamic translator, this is referred to as *context switch*. Translated instructions are held through the **translation cache** component. Basically it is contiguous region of memory reserved at the startup of the dynamic translator where all translated blocks are stored. This area of memory must be marked as executable as the translated code is executed in place once the translation is completed. In order to have a good cache performance the translated block should be correctly aligned and allocated trying to enhance the locality of the code. When no more space is available in the cache for new fragments, the entire content of the cache is discarded. This is the cache management policy used by the Dynamo dynamic optimizer [4]. The advantages of this method are simplicity of implementation and fast fragment allocation and deallocation. The disadvantage is that when the executed portions of the application binary do not entirely fit within the cache, the simple eviction policy results in potentially useful fragments being discarded.

The translating process starts capturing the application context, and searching for a translation for the block of source code which begins at the address pointed by the EIP register. If a translation for this instruction is found in the cache, a context switch restores the application context and starts executing the translated version. If there is no translation in the cache, the space for a new fragment is allocated in the translation cache and the control is passed to the **translation unit**.

The task of the translation unit is to performs an application-specific translation and guarantee the correctness of the application. It starts fetching instructions from the address in the EIP register saved within the application context, until it reaches a unconditional branch instruction or the maximum number of instruction per block.Than it decodes the instructions to be able to analyse them. It is particular interesting the approach taken by DynamoRio, which has developed an adaptive level-of-detail representation algorithm for decoding the instructions, where an instruction is only decoded and encoded as much as is needed. Note that there is not need to use a code discovery algorithm as in static approach because a fragment consists exclusively of instructions. Once the instructions are correctly translated, they can be analysed for searching for instructions of interest. In the case of the system call interceptor the translation unit should search for instructions that might issue a system call request presented in 4. If one of those instructions is found, this must be overwritten with a jump to a system-call handling routine. Another advantage of performing binary rewriting at run time is that an instruction can always be substituted with another instruction even though the instruction set is of variable length. The translated instructions are copied in a new empty frame into the translation cache, therefore there will always be enough space to allocate a jump instruction.

During the translation process, the operand of a branch instruction must be adjusted to point to the translated block within the translation cache. This is ensue that only translated instruction are executed and therefore no system call invocation is missed. Usually the address are translated using

a fast look-up hash table. If a translation for the target instruction does not exist, then the translation unit is invoked to translate the target block and insert it in the cache unit. Note that this approach might lead to duplicate code in the cache unit, for example when the operand of jump is an address pointing to an instruction in the middle of an block which has been already translated. A method used to improve the performance of handling branch instructions is to patch direct jumps and direct calls to avoid future lookups, this has been used in [13, 4]. Unfortunately, patching cannot be used for indirect branches, including indirect calls and returns as their targets are variable. This hash table lookup for indirect branches, especially during return instructions, has been reported in [13] to be the main source of slowdown in the execution of the dynamic rewritten binary. When the translation of a fragment is terminated, a context switch is performed and the application starts executing that fragment. The translation process is depicted in

In the research literature, there are two examples of system call interceptor that are implemented following the approach previously discussed. The first is called Strata ,described in [39], where a system call interceptor has been used to ensure a safe virtual execution for an untrusted binary. A 40% of overhead is introduce in the execution of binary using the Strata tool, this is due principally to the rewriting process. Strata has been implemented by following an easy approach concerning more about portability rather than performance and no optimizations have been introduced. A better and more complex example is implemented in vx32 [13]. Vx32 is a lightweight sandbox that relies on segment protection as well as binary translation to confine an untrusted application.Vx32 is provided as static library that can be used to implement security tools. *VxLinux* is a small system call interceptor implemented using vx32 library, it can be found at [12]. The interceptor mechanism is implemented following a delegating approach similar to that describe in chapter 3.3, it is able to run unmodified binary introducing an overhead spanning from 10% to 50% worst case.

4.3 Seccomp

A system call interceptor has been often employed in building sandboxing tools [43, 15, 2, **?**].The crucial aspect of these tools is security as a sandbox must guarantee that all system calls made by the sandboxed processes are correctly intercepted and verified. Current interposition-based mechanisms offer a wide variety of properties that make them an attractive approach for building sandboxing systems. Unfortunately, these are affects by numerous shortcomings that make their implementation difficult and a possible source of vulnerabilities. For instance, independently from the interceptor mechanism used, a sandbox has to deal with different type of race conditions such as arguments races, relative path races and symbolic link race, for a comprehensive list see [14, 46]. These are sometimes referred to as *Time of Check to Time of Use (TOCTOU)* races. In general, a race condition arises when there is a shared resource among different threads and the following actions occur :

1. A sandbox grants permission to perform an operation A, that relies on some mutable shared state. For example, a shared buffer containing the name of a file is the arguments of a system call.

2. Before the operating system starts performing the request, the shared resource is changed by another thread, making the operation A illegal. The sandbox will not intercept this action because a thread can modified a shared buffer without invoking a system call as the shared buffer is located in its own space memory.

3. The operation A is performed by the operating system on the illegal values.

This type of race condition is a significant problem for sandboxing tools and can be used to mislead introduction detections system and perform not authorized operations. In order to tackle these problems and simplify the implementation of sandboxing tools, a simple sandboxing mechanism called **seccomp** (short for secure computing mode) has been introduced in Linux kernel 2.6.12. Seccomp isolates the execution of an untrusted process in its own address space by limiting the available system calls that the untrusted process can invoke. Seccomp mode can be enabled via the `prctl` [26] system call using `PR_SET_SECCOMP` as first argument, when the system call returns, the caller

process enters seccomp mode. Seccomp has two different working modalities that can be selected via its second arguments:

SECCOMP_MODE_STRICT sets the most restrictive working mode where the only system calls that the thread is permitted to make are read, write, exit, an sigreturn. Other system calls result in the delivery of a SIGKILL signal that terminates the application's execution. Strict secure computing mode is useful for applications that may need to execute untrusted byte code obtained from an external source, for example, by reading from a pipe or socket. Seccomp was first devised by Andrea Arcangeli in January 2005 for safely running numerical applications in public grid computing. It has also been used by Google to build a sandbox environment within Chrome called *seccompsandbox* [16]. This is used to restrict the execution of Adobe Flash Player and video renderers.

SECCOMP_MODE_FILTER has been recently introduced in Linux 3.5. This approach provides more flexibility respect the strict mode as it allows to define which system call should be denied via a Berkeley Packet Filter [29] passed as third argument.This can be designed to filter arbitrary system calls and system call arguments. This approach has been used to implement *minijail* [17] an new sandbox used in Chrome and in VSFTP [42].

In the remainder of this section the two seccom modes are analysed in better details. The section 4.3.1 introduces *seccompsandbox*, the sandbox implemented in Chromium which uses a selective dynamic binary rewriting approach to implement a delegating system call interposition. While 4.3.2 present the features provided by the filtering mode of seccomp.

4.3.1 Seccomp mode strict

Creating a sandbox in which to run untrusted code is a difficult problem. The successful sandbox implementations tend to come with completely new languages (e.g. Java) that are specifically designed to support that functionality. Try to sandbox an application at system call level as in Janus [43], Ostia [15] is a much more difficult task due principally to the problems discussed in the previous section. Seccomp offers a perfect solution for implementing a sandbox because it reduces the kernel attack surface to just four system calls. This ensures that, even though the application is compromise, an attacker cannot perform any permanent damage in the system. The only system calls allowed in seccom strict mode are :

- read/write can be used to write to or read from a file descriptor. Note that the file descriptor must be already open.
- sigreturn is necessary to support correctly signal in Linux. This system call is called every time the signal handler has completed its task in order to return the control to the original process.
- exit is used to terminate the process.

If a process attempts to call any other system call, the entire process is terminated. This behaviour is very desirable from a security point of view, as it means that any failure in the sandbox causes the termination of the sandboxed process preventing possible unsafe actions.Seccomp was originally designed to isolate number-crunching applications in grid computing. Numerical applications usually do not require a large amount of system calls and thus they can be successfully executed just with these few system calls. However, this is not true in general; the four system calls allowed by seccomp are insufficient for most applications to run successfully.

To overcome this limitation both *seccompsandbox* and *seccomp-nurse* run a trusted helper thread that does not enable seccomp. The helper thread opens a communication channel with the untrusted thread (e.g. using a socketpair or pipe). Now, any time the sandboxed thread wants to make a system call other than one of the four unrestricted system calls, it serializes the request and writes it over the communication channel with the helper thread.The helper then inspects the request and if it is a

valid request, it executes it on behalf of the sandboxed thread.This architecture has been depicted in Figure 4.2.

This delegating architecture works for correctly most system calls, but a small number of system calls that manipulate the thread state need to be emulated differently. Notable and relevant examples are thread-local storage, and POSIX signals. Fortunately, TLS is a non-issue as it gets set up once at thread creation and is then not touched any more.Signals are more difficult and they are emulated in user-space. For an application that makes heavy use of signals, this could turn out to be a problem.

One of the requirement of a sandbox is to run untrusted binaries without requiring sources modifications and recompilation. In order to satisfy this requirement the delegating architecture presented so far must solve two problems.

- The first is how to insert the untrusted application in seccomp mode. The request to enter seccomp mode must be issued by the untrusted application, but none application has been natively designed to work on seccomp mode.
- The second problem is how to prevent the untrusted process from issuing system calls. All system call requests should be intercepted and routed to the helper thread before the operating system start processing them, otherwise the application will be terminated.

In order to insert the untrusted application in seccomp, a prtcl request must be inject into the untrusted application before its execution starts. This can be accomplished in different ways, for example by tracing the untrusted process and substituting its first system call request with a prctl request. While, seccompsandbox and seccomp-nurse use an approach based on LD_PRELOAD[4] feature to solve this problem.

The entry point of an executable file is the *_start* routine provided by the compiler. The main task of this routine is to initialize the stack and call the `__libc_start_main` provided by the GNU libc. The `__libc_start_main` routine performs some initialisation actions to set up the execution environment and then runs the application by calling its main function. The LD_PRELOAD feature can be used to override the `__libc_start_main` function in order to call an arbitrary function. Seccomp-nurse uses this mechanism to initialize the sandbox environment and inject the prtcl request. The `__libc_start_main` is overridden by the function presented in Listing 4.3.1 during the load phase. This function retrieves the address of the original `__libc_start_main` function from the GUN libc library (lines) and then calls it by replacing the original main by a modified version called `wrap_main`. This causes the execution environment to be properly initialised and the wrapper function to be executed. The aim of this wrapper function is to perform all those essential actions to correctly set up the sandbox environment, for example installing the communication channel with the helper thread.

[4]LD_PRELOAD is a environment variable that contains a location of a shared object. During the linking phase this object will be linked to the binary allowing for symbols to be overwritten.

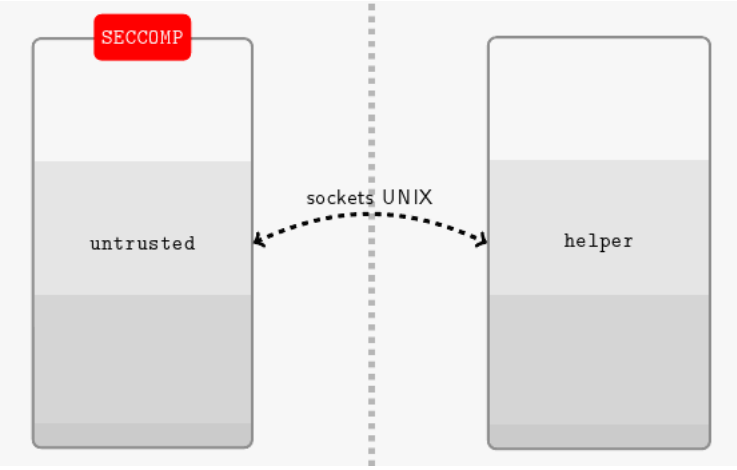

Figure 4.2: Seccomp delegating architecture

Once the initialisation phase is completed, a prctl request is issued and the application enters strict seccomp mode execution.

```
typedef int (*main_t)(int, char **, char **);
main_t realmain;

int __libc_start_main(main_t main,
                      int argc,
                      char *__unbounded *__unbounded ubp_av,
                      ElfW(auxv_t) *__unbounded auxvec,
                      __typeof (main) init,
                      void (*fini) (void),
                      void (*rtld_fini) (void), void *__unbounded
                      stack_end)
{
        void *libc;
        int (*libc_start_main)(main_t main,
                               int,
                               char *__unbounded *__unbounded,
                               ElfW(auxv_t) *,
                               __typeof (main),
                               void (*fini) (void),
                               void (*rtld_fini) (void),
                               void *__unbounded stack_end);

        libc = dlopen("libc.so.6", RTLD_LOCAL  | RTLD_LAZY);
        if (!libc)
                ERROR("  dlopen() failed: %s\n", dlerror());
        libc_start_main = dlsym(libc, "__libc_start_main");
        if (!libc_start_main)
                ERROR("     Failed: %s\n", dlerror());

        realmain = main;
        return (*libc_start_main)(wrap_main, argc, ubp_av, auxvec,
        init, fini, rtld_fini, stack_end);

    int wrap_main(int argc, char **argv, char **environ)
{
    if (socketpair(AF_UNIX, SOCK_STREAM, 0, fds) < 0) {
          perror("socketpair failed");
                exit(1);

    }

        if (prctl(PR_SET_SECCOMP, 1, 0, 0) == -1) {
                perror("prctl(PR_SET_SECCOMP) failed");
                printf("Maybe you don't have the CONFIG_SECCOMP support built
                   into your kernel?\n");
                exit(1);
        }

        (*realmain)(argc, argv, environ);
}
}
```

Listing 4.6: Wrapper __libc_start_main used in seccomp-nurse

Note that this method is affected by a serious vulnerability. A binary file can be generate in such way that the *_start* routine does not call the `__libc_start_main`. In that case, the prtcl routine would not be called and the program would not run in a sandboxed environment. Seccompsandbox uses a

similar approach, it loads am initialization function in memory using LD_PRELOAD but it redirect the execution to this function using ptrace (i.e. it explicitly sets the pointer register to the entry within the object load via LD_PRELOAD). This approach avoids the vulnerability previously discussed.

Once the untrusted process has entered seccomp mode process, its attempts to invoke a system call must be intercepted before the operating system starts processing them and redirect to the helper thread. The helper thread then can thoroughly verify the system call and its arguments and decide whether execute or deny it. Seccompsandbox uses a selective dynamic rewriter to intercept system call invocations made by the untrusted thread and the libraries to which it has been linked, this process is usually referred to as patching phase. The patching phase occurs during the initialization phase before the application enters seccomp mode.

Seccompsandbox uses the rewriting method presented in BIRD [30] in a dynamic manner. The first difference between seccompsandbox and the binary methods previously discussed is that seccombsandbox reduces the discovery code phase to merely disassembling the application code segment. This implementation choice has been taken because it does not compromise the security of the sandbox as any failure to find the system call sites or any failure to correctly rewrite them would result in the termination of the application.

System call instructions are rewritten with an unconditional jump instruction. During the rewriting process, some instructions nearby the system call instruction may be relocate to provide enough space to locate the jump instruction. The target of the jump instruction is a routine that is dynamically allocated during the rewriting process via memmap system call. This routine accomplishes the following tasks in order to ensure that the application's behaviour has not been changed during the rewriting process :

- Executes the instructions that have been relocated during the rewriting process by guaranteeing the original order. Usually these instructions are referred to as preamble and postable.
- Performs a translation between the system call convention (system call arguments within the registers) to the C function convention (functions arguments in the stack) and call a dispatch routine which routes the system call invocation to the correct system call handler.
- Ensures that the rewriting process is fully transparent to the untrusted thread. This accomplished by saving the state before calling the dispatcher routine ans restoring it with the exception of the register that contains the result of the system call.

Once the patching phase is completed, the system call requests are routed to the helper thread using the communication channel installed during the initialisation phase. This method and more in general a delegating architecture, requires to implement a dedicate handler for almost all system calls implemented in Linux. This is not a trivial implementation effort due principally to the high number of system calls (over 300) and their complexity. Some system calls, for example signal handling,memory mapping and process management, cannot be just executed on behalf of the untrusted thread as these system call may change the thread state. The approach followed to address this problem usually consist of emulating kernel functionalities at user space. However, in some case this is not possible as in the case of the clone system call and mmap. This is one of the the main reason that has driven to develop an alternative solution.

An exemple of this routine for x86_32 architecture is presented in .

```
execute_preable()
salve_state();
// transform system call invocation to the c convention
push %edi
  push %esi
  push %edx
  push %ecx
  push %ebx
  push %eax
```

```
    // Call default handler.
  "call playground$defaultSystemCallHandler@PLT\n"
  //clean the stack
  "add  $24, %esp\n"
    restore_state();
    execute_
//
```

Listing 4.7: Synopsis utrace_set_events_task

4.3.2 Seccomp mode filter

A large number of system calls are exposed to every userland process with many of them going unused for the entire lifetime of the process. As system calls change and mature, bugs are found and eradicated. A certain subset of userland applications benefit by having a reduced set of available system calls. The resulting set reduces the total kernel surface exposed to the application. System call filtering is meant for use with those applications.

Seccomp filtering provides a means for a process to specify a filter for incoming system calls. The filter is expressed as a Berkeley Packet Filter (BPF) program, as with socket filters, except that the data operated on is related to the system call being made: system call number and the system call arguments. This allows for expressive filtering of system calls using a filter program language with a long history of being exposed to userland and a straightforward data set.

Additionally, BPF makes it impossible for users of seccomp to fall prey to time-of-check-time-of-use (TOCTOU) attacks that are common in system call interposition frameworks. BPF programs may not dereference pointers which constrains all filters to solely evaluating the system call arguments directly.

If `SECCOMP_MODE_FILTER` filters permit fork(2), then the seccomp mode is inherited by children created by fork(2); if execve(2) is permitted, then the seccomp mode is preserved across execve(2). If the filters permit prctl() calls, then additional filters can be added; they are run in order until the first non-allow result is seen.

Appendix A

Appendix Linux System Call

In this section we introduce how system calls are implemented in Linux and at least the ABI convention used to pass parameters across different functions.

Appendix B

Executable and Linkable Format

ELF

Bibliography

[1]

[2] Anurag Acharya, Mandar Raje, and Ar Raje. Mapbox: Using parameterized behavior classes to confine applications. In *In Proceedings of the 9th USENIX Security Symposium*, pages 1–17, 2000.

[3] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[4] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent dynamic instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, VEE '12, pages 133–144, New York, NY, USA, 2012. ACM.

[5] Petr Hosek Cristian Cadar. Multi-version software updates. In *Workshop on Hot Topics in Software Upgrades (HotSWUp 2012),*, 6 2012.

[6] Renzo Davoli. Kmview @ONLINE, October 2007.

[7] Mathieu Desnoyers and Michel R. Dagenais. The lttng tracer: A low impact performance andbehavior monitor for gnu/linux. 2006.

[8] Je Dike. Linux as a hyper visor. 2001.

[9] Je Dike. User-mode linux. 2001.

[10] Kemal Ebcioglu and Erik R. Altman. Daisy: Dynamic compilation for 1001997.

[11] Jake Edge. A /proc/pid/mem vulnerability. `http://lwn.net/Articles/476947/`, 2012.

[12] Bryan Ford and Russ Cox. Vx32, 2008.

[13] Bryan Ford and Russ Cox. Vx32: lightweight user-level sandboxing on the x86. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 293–306, Berkeley, CA, USA, 2008. USENIX Association.

[14] Tal Garfinkel. Traps and pitfalls: Practical problems in in system call interposition based security tools. In *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[15] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *IN NDSS*, 2003.

[16] Google. Seccompsandbox, 2010.

[17] Google. minijail, 2013.

[18] Philip J. Guo and Dawson Engler. Cde: Using system call interposition to automatically create portable software packages. 2011.

[19] Tejun Heo. Linux mailing list. `http://thread.gmane.org/gmane.linux.kernel/1107045`, 2011.

[20] Andrew P. Kosoresow and Steven A. Hofmeyr. Intrusion detection via system call traces. *IEEE Software*, 14:35–42, 1997.

[21] Paul Kranenburg. Strace. `http://sourceforge.net/projects/strace/`, 2013.

[22] Michael Laurenzano, Mustafa M. Tikir, Laura Carrington, and Allan Snavely. Pebil: Efficient static binary instrumentation for linux. In *ISPASS*, pages 175–183. IEEE Computer Society, 2010.

[23] Federico Maggi, Matteo Matteucci, and Stefano Zanero. Detecting intrusions through system call sequence and argument analysis. *IEEE Transactions on Dependable and Secure Computing*, 7(4):381–395, 2010.

[24] Linux man-page Project. Linux programmer's manual clone. `http://man7.org/linux/man-pages/man2/__clone2.2.html`, 2013.

[25] Linux man-page Project. Linux programmer's manual cross memory attach. `http://man7.org/linux/man-pages/man2/process_vm_readv.2.html`, 2013.

[26] Linux man-page Project. Linux programmer's manual prctl. `http://man7.org/linux/man-pages/man2/prctl.2.html`, 2013.

[27] Linux man-page Project. Linux programmer's manual ptrace. `http://man7.org/linux/man-pages/man2/ptrace.2.html`, 2013.

[28] Ananth Mavinakayanahalli, Prasanna Panchamukhi, and Jim Keniston. Probing the guts of kprobes. 2006.

[29] Steven Mccanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. pages 259–269, 1992.

[30] Susanta Nanda, Wei Li, Lap chung Lam, and Tzi cker Chiueh. Bird: Binary interpretation using runtime disassembly.

[31] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[32] Cristian Cadar Petr Hosek. Safe software updates via multi-version execution. In *International Conference on Software Engineering (ICSE 2013)*, 5 2013.

[33] DjprobeKernel probing with the smallest overhead. Masami hiramatsu and satoshi oshima. 2007.

[34] Niels Provos. Improving host security with system call policies. In *In Proceedings of the 12th Usenix Security Symposium*, pages 257–272, 2002.

[35] Ocvtavian Purdila and Andreas Terzis. A dynamic browser containment environment for countering web-base malware. 2006.

[36] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. Instrumentation and optimization of win32/intel executables using etch. In *In Proceedings of the USENIX Windows NT Workshop*, pages 1–7, 1997.

[37] Yasushi Saito. Jockey: A user-space library for record-replay debugging. In *In AADEBUG05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 69–76. ACM Press, 2005.

[38] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: Intrusion detection using parallel execution and monitoring of program variants in userspace. In *In Proceedings of the European Conference on Computer Systems*, 2009.

[39] Kevin Scott and Jack Davidson. Safe virtual execution using software dynamic translation. In *Annual Computer Security Applications Conference*, pages 209–218, 2002.

[40] Mark Seaborn. Plashglibc, 2008.

[41] Goswami Sudhanshu. An introduction to kprobes @ONLINE, 2005.

[42] vsftp. vsftp, 2013.

[43] David A. Wagner. Janus: an approach for confinement of untrusted application.

[44] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, SOSP '93, pages 203–216, New York, NY, USA, 1993. ACM.

[45] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 299–308, New York, NY, USA, 2012. ACM.

[46] Robert N. M. Watson. Exploiting concurrency vulnerabilities in system call wrappers.

[47] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *In Measurement and Modeling of Computer Systems*, pages 68–79, 1996.

[48] C. P. Wright, R. Spillane, G. Sivathanu, and E. Zadok. Extending acid semantics to the file system. *ACM Transactions on Storage (TOS)*, 3(2):1–42, June 2007.

[49] Guido Van t Noordende, dm Balogh, Rutger Hofman, Frances M. T. Brazier, and Andrew S. Tanenbaum. A secure jailing system for confining untrusted applications. In *International Conference on Security and Cryptography (SECRYPT*, pages 28–31.