System call tracing overhead

Jörg Zinke

Potsdam University
Institute for Computer Science
Operating Systems and Distributed Systems

Dresden, 2009/10/29

Outline

Introduction

- system call tracing is a common used technique for debuggers or applications which enforce security policies
- for debugging purposes the tracing is often done *step-by-step* or in conjunction with breakpoints
- for enforcing security policies usually the interception of system calls is required

  $\longrightarrow$ modify, forbid or allow system calls

Introduction

- system call tracing is a common used technique for debuggers or applications which enforce security policies
- for debugging purposes the tracing is often done *step-by-step* or in conjunction with breakpoints
- for enforcing security policies usually the interception of system calls is required
  - $\longrightarrow$ modify, forbid or allow system calls

Overview

- system call interception requires at least a kernel based implementation and an user space process to trace
- usually, there is another process which triggers the tracing and does some actions before and maybe after the system call
- in addition the triggering requires some kind of registration at the kernel implementation, at least the PID of the application to trace is required
- commonly, kernel implementations provide mechanisms for reading processor registers
  - → gives the possibility to modify arguments or even modify the data pointed to by arguments

Overview

- system call interception requires at least a kernel based implementation and an user space process to trace
- usually, there is another process which triggers the tracing and does some actions before and maybe after the system call
- in addition the triggering requires some kind of registration at the kernel implementation, at least the PID of the application to trace is required
- commonly, kernel implementations provide mechanisms for reading processor registers
  - → gives the possibility to modify arguments or even modify the data pointed to by arguments

Overhead

- intercepting system calls involves additional overhead
- additional overhead can be ignored for the purpose of debugging *but* should be considered for security enforcing applications and other kind of applications

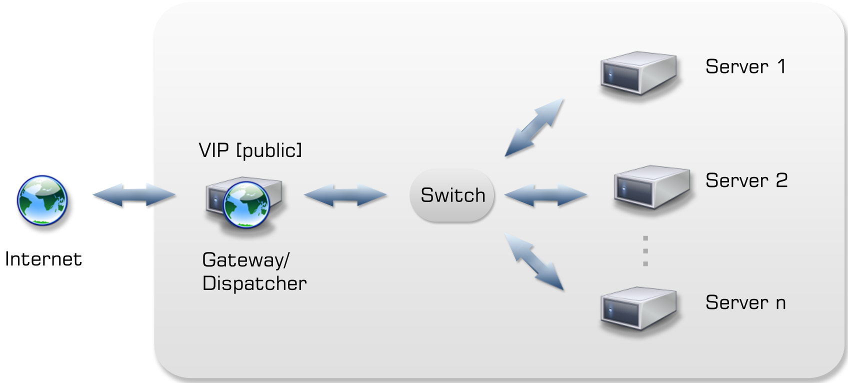
$\longrightarrow$ determine the additional overhead through measurements

Overhead

- intercepting system calls involves additional overhead
- additional overhead can be ignored for the purpose of debugging *but* should be considered for security enforcing applications and other kind of applications

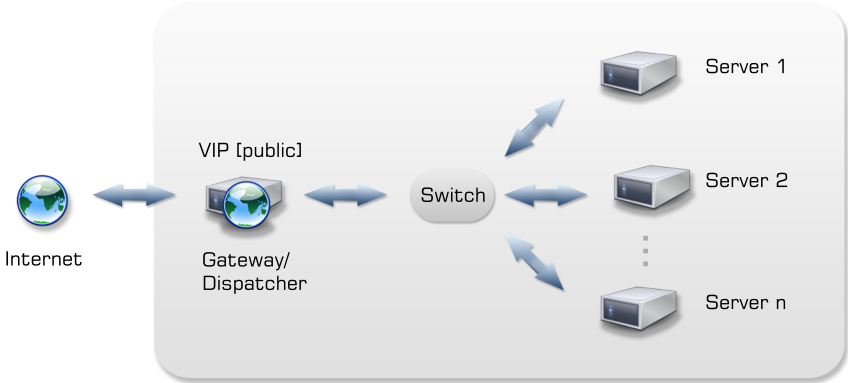
$\longrightarrow$ determine the additional overhead through measurements

Background: server load balancing



→ trace socket system calls of processes on backend servers to determine useful metrics and values for load balancing

Background: server load balancing



→ trace socket system calls of processes on backend servers to determine useful metrics and values for load balancing

System call

- interface between operating system kernel and user space programs is defined by a set of system calls [Tan01]
- system calls vary from OS to OS but concepts tend to be similar
- system calls transfers the control to the OS similar to a function call which enters the kernel

→ system calls are a universal and fundamental mechanism

System call

- interface between operating system kernel and user space programs is defined by a set of system calls [Tan01]
- system calls vary from OS to OS but concepts tend to be similar
- system calls transfers the control to the OS similar to a function call which enters the kernel

→ system calls are a universal and fundamental mechanism

System call implementation

- system call tracing implementations are usually use system calls too, for registering of PIDs from user space
- implementing a system call requires control transfer, often done through interrupts or traps
- modern architectures provide `SYSCALL/SYSRET` or `SYSENTER/SYSEXIT` instructions for fast control transfer [BC05]
- system call implementations have to take care about restricted rights and access control
  - → e.g. `open()` has to check whether the file permissions and the owner match the issuing process

System call implementation

- system call tracing implementations are usually use system calls too, for registering of PIDs from user space
- implementing a system call requires control transfer, often done through interrupts or traps
- modern architectures provide `SYSCALL/SYSRET` or `SYSENTER/SYSEXIT` instructions for fast control transfer [BC05]
- system call implementations have to take care about restricted rights and access control
  - → e.g. `open()` has to check whether the file permissions and the owner match the issuing process

System call interception

Three approaches for system call interception mentioned in [Pet97]:

1. kernel based system call interception implemented through a modified system kernel
2. using a modified system library to *replace* the default system calls (maybe using shared libraries and *preload* mechanisms)
3. using a trace process and a debugging interface like ptrace or systrace for system call interception of applications

Focus

→ focus on stable implementations of the third approach in standard kernels on Linux and OpenBSD, namely ptrace and systrace

→ microbenchmarks to determine overhead, issued through context switches between traced process and application

→ kernel tracer like ktrace are out of scope caused by background of server load balancing

Focus

- → focus on stable implementations of the third approach in standard kernels on Linux and OpenBSD, namely ptrace and systrace
- → microbenchmarks to determine overhead, issued through context switches between traced process and application
- → kernel tracer like ktrace are out of scope caused by background of server load balancing

Focus

- → focus on stable implementations of the third approach in standard kernels on Linux and OpenBSD, namely ptrace and systrace
- → microbenchmarks to determine overhead, issued through context switches between traced process and application
- → kernel tracer like ktrace are out of scope caused by background of server load balancing

Focus

→ focus on stable implementations of the third approach in standard kernels on Linux and OpenBSD, namely ptrace and systrace

→ microbenchmarks to determine overhead, issued through context switches between traced process and application

→ kernel tracer like ktrace are out of scope caused by background of server load balancing

Debugging applications using system call tracing

- well known GDB
- ftrace based on frysk
- DTrace on Solaris
- strace based on ptrace
- truss (FreeBSD/SunOS/System V)

→ various commonly used applications for debugging and tracing other processes are available

Debugging applications using system call tracing

- well known GDB
- ftrace based on frysk
- DTrace on Solaris
- strace based on ptrace
- truss (FreeBSD/SunOS/System V)

→ various commonly used applications for debugging and tracing other processes are available

Security applications using system call tracing

- AppArmor
- SELinux
- grsecurity
- systrace

→ various commonly used applications to limit application access and achieve Mandatory Access Control (MAC) are available

Security applications using system call tracing

- AppArmor
- SELinux
- grsecurity
- systrace

$\longrightarrow$ various commonly used applications to limit application access and achieve Mandatory Access Control (MAC) are available

Performance system call interception

The performance overhead of a system call interception can be split into the following two parts [Chu]:

- cost of system call interception, for example passing control to the tracing process at every system call of the traced process
- cost of the analysis performed by the tracing process, for example determining whether the reported request for kernel service should be allowed every time the tracing process get invoked

Performance studies systrace

Command	Real	User	System
`find /usr/src/ >/dev/null`	30	0.2	0.3
`systrace find /usr/src/ >/dev/null`	42	1.2	3.8
`gzip -9 test.bin`	2.0	1.7	0.1
`systrace gzip -9 test.bin`	1.9	1.6	0.1

Table: Systrace overhead from [Pro02].

Other approaches

- reducing the overhead of ptrace through subsets of system calls (policies)
- utrace and uprobe as replacement for ptrace
- lbox framework (more efficient than ptrace or systrace)

$\longrightarrow$ shortcomings in performance and overhead of ptrace and systrace require maybe completely different approaches

Other approaches

- reducing the overhead of ptrace through subsets of system calls (policies)
- utrace and uprobe as replacement for ptrace
- lbox framework (more efficient than ptrace or systrace)

$\longrightarrow$ shortcomings in performance and overhead of ptrace and systrace require maybe completely different approaches

Ptrace

- ptrace system call and kernel implementation is available on Linux and OpenBSD and on various further operating systems
- the system call itself is not part of the POSIX standard
- both implementations, Linux and OpenBSD are similar, but the OpenBSD implementations lacks support of features like `PTRACE_SYSCALL` (only `PTRACE_SINGLESTEP`)

→ only the Linux implementation is considered in the following

Ptrace

- ptrace system call and kernel implementation is available on Linux and OpenBSD and on various further operating systems
- the system call itself is not part of the POSIX standard
- both implementations, Linux and OpenBSD are similar, but the OpenBSD implementations lacks support of features like `PTRACE_SYSCALL` (only `PTRACE_SINGLESTEP`)

→ only the Linux implementation is considered in the following

Ptrace sequence

- attach and detach via PTRACE_ATTACH and PTRACE_DETACH system call arguments
- attach means: tracing application becomes parent of the traced process
- detach restores original parent

→ main idea is: attach to another process identified by PID, start tracing and detach later

Ptrace sequence

- attach and detach via `PTRACE_ATTACH` and `PTRACE_DETACH` system call arguments
- attach means: tracing application becomes parent of the traced process
- detach restores original parent

→ main idea is: attach to another process identified by PID, start tracing and detach later

Ptrace capabilities and options

- traced process stops on monitored event (system calls or single step) and sends `SIGCHLD` signal its parent
- a process can not be traced by two processes at the same time
- `CAP_SYS_PTRACE` capability flag is required to trace every process in system except `init`
- without capability flag set only processes of the same owner are allowed to trace

→ tracing parent can read registers and data from the *stopped* traced process memory

Ptrace capabilities and options

- traced process stops on monitored event (system calls or single step) and sends `SIGCHLD` signal its parent
- a process can not be traced by two processes at the same time
- `CAP_SYS_PTRACE` capability flag is required to trace every process in system except `init`
- without capability flag set only processes of the same owner are allowed to trace

→ tracing parent can read registers and data from the *stopped* traced process memory

Ptrace drawbacks

- ptrace does not allow monitoring of *specific* system calls, instead just *all* system calls are monitored
  - $\longrightarrow$ incurring at least two context switches per traced system call
- blocks the traced process on every system call it makes, therefore the tracing process needs to continue the child each time it is blocked
- considering that a tracing process might monitor more than one process, the overhead on the tracing process increases

Systrace

- developed by Niels Provos
- term systrace refers to the application as well as to the system call and the according kernel implementation
- available for various operating systems, uses different kernel implementations depending on the operating system, for example, the systrace application uses ptrace on Linux

→ focus on the kernel based systrace implementation which is available in the OpenBSD Kernel

Systrace

- developed by Niels Provos
- term systrace refers to the application as well as to the system call and the according kernel implementation
- available for various operating systems, uses different kernel implementations depending on the operating system, for example, the systrace application uses ptrace on Linux

→ focus on the kernel based systrace implementation which is available in the OpenBSD Kernel

Systrace sequence

- enforce policies on system calls
- user space process controls behavior through pseudo-device `/dev/systrace` and an `ioctl` based interface
- the `ioctl` interface together with the defined systrace messages achieve various tracing operations (similar to ptrace), like `STRIOCIO` for copying data in/out of the process being traced

→ systrace attaches to another process identified by PID first, start tracing them and detach later (similar to ptrace)

Systrace sequence

- enforce policies on system calls
- user space process controls behavior through pseudo-device `/dev/systrace` and an `ioctl` based interface
- the `ioctl` interface together with the defined systrace messages achieve various tracing operations (similar to ptrace), like `STRIOCIO` for copying data in/out of the process being traced

→ systrace attaches to another process identified by PID first, start tracing them and detach later (similar to ptrace)

Systrace policies

- three policies can be assigned to system calls
  - SYSTR_POLICY_PERMIT - immediately allow the system call
  - SYSTR_POLICY_NEVER - forbids the system call
  - SYSTR_POLICY_ASK - sends a message of the type SYSTR_MSG_ASK and puts the process to sleep until the according answer

→ besides the flexibility of systrace policies, they should be fast since basic policies SYSTR_POLICY_PERMIT and SYSTR_POLICY_NEVER are handled in kernel without asking user space (fast path)

Systrace policies

- three policies can be assigned to system calls
  - `SYSTR_POLICY_PERMIT` - immediately allow the system call
  - `SYSTR_POLICY_NEVER` - forbids the system call
  - `SYSTR_POLICY_ASK` - sends a message of the type `SYSTR_MSG_ASK` and puts the process to sleep until the according answer

→ besides the flexibility of systrace policies, they should be fast since basic policies `SYSTR_POLICY_PERMIT` and `SYSTR_POLICY_NEVER` are handled in kernel without asking user space (fast path)

Measurement environment

- Dual Core Xeon 1.86 GHz in a dual-boot configuration running CentOS 5.2 for ptrace measurements and OpenBSD 4.3 for the systrace measurements
- all measurements gather the number of CPU cycles through the `rdtsc` register and all are repeated 51 times explictly to avoid cache effects
- the median is used to calculate the result from the 51 repetitions to obviate distortions

Measurement experiments

- first the number of CPU Cycles for the three single plain system calls `open()`, `write()` and `close()` are measured
- then the measurements are done again while tracing these system calls
- additional measurements are done for all three system calls in a sequence and again with an invalid file descriptor

Plain system calls

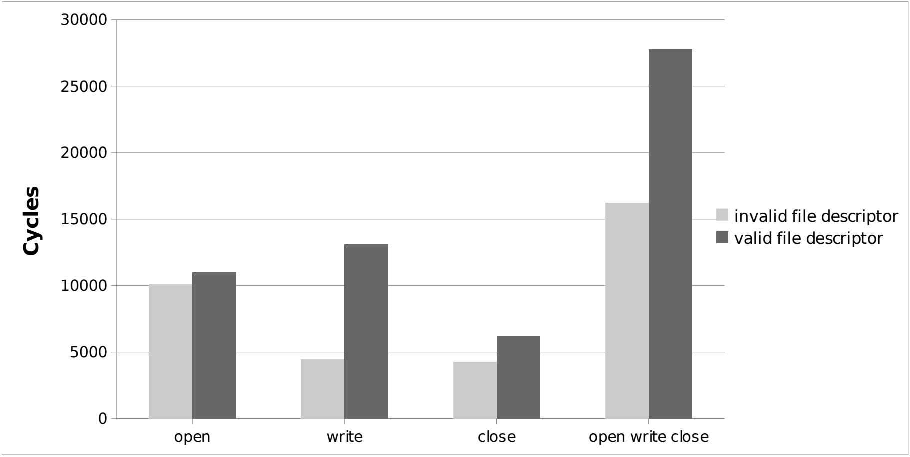


Figure: System calls with and without valid file descriptors.

Flapping effect

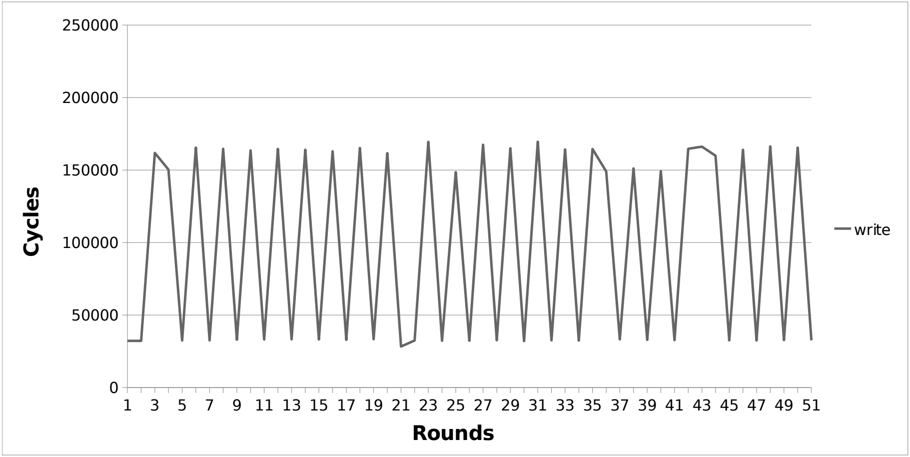


Figure: Flapping for `write()` and ptrace with invalid file descriptor.

Overhead with valid file descriptors

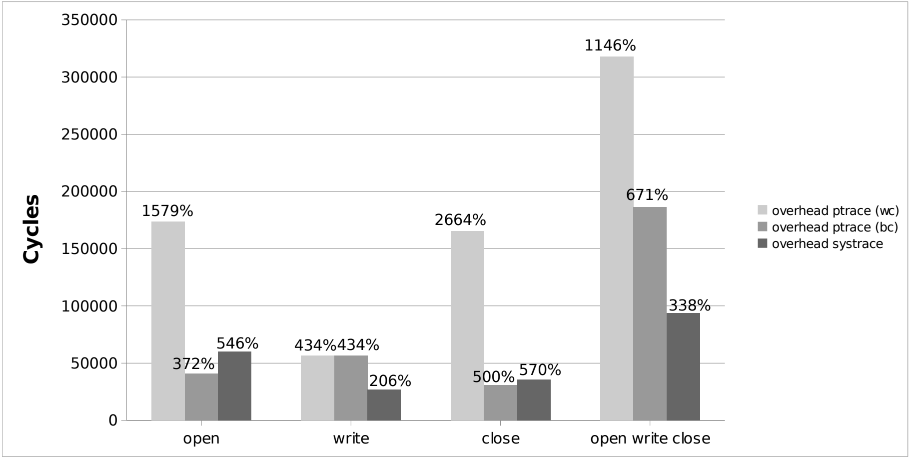


Figure: Overhead for ptrace and systrace with valid file descriptors.

Overhead with invalid file descripors

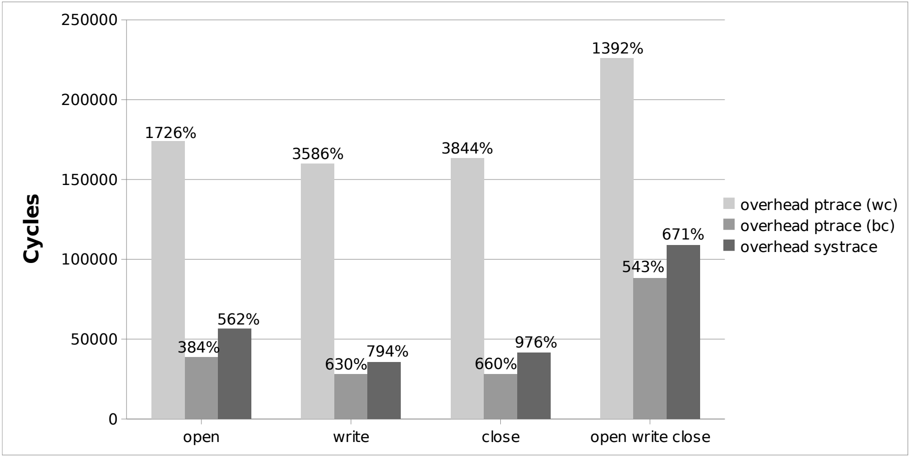


Figure: Overhead for ptrace and systrace with invalid file descriptors.

Conclusion and future work

- system call interception through systrace or ptrace is considered as slow
- a sequence of system calls is faster than all involved single system calls
- ptrace interface measurements show some strange flapping results, which are in the worst case scenario slower then the competitor measurements from systrace
- policy concept of systrace is considered to be faster and more flexible than ptrace
- overhead looks dramatically high

→ can be considered as negligible small for the sake of improved security and flexibility

→ further macrobenchmarks measurements in the field of self-adapting server load balancing have shown that the overhead can not be considered as negligible small therefore

Conclusion and future work

- system call interception through systrace or ptrace is considered as slow
- a sequence of system calls is faster than all involved single system calls
- ptrace interface measurements show some strange flapping results, which are in the worst case scenario slower then the competitor measurements from systrace
- policy concept of systrace is considered to be faster and more flexible than ptrace
- overhead looks dramatically high

→ can be considered as negligible small for the sake of improved security and flexibility

→ further macrobenchmarks measurements in the field of self-adapting server load balancing have shown that the overhead can not be considered as negligible small therefore

Conclusion and future work

- system call interception through systrace or ptrace is considered as slow
- a sequence of system calls is faster than all involved single system calls
- ptrace interface measurements show some strange flapping results, which are in the worst case scenario slower then the competitor measurements from systrace
- policy concept of systrace is considered to be faster and more flexible than ptrace
- overhead looks dramatically high

→ can be considered as negligible small for the sake of improved security and flexibility

→ further macrobenchmarks measurements in the field of self-adapting server load balancing have shown that the overhead can not be considered as negligible small therefore

Literature I

Daniel Bovet and Marco Cesati.
Understanding The Linux Kernel.
Oreilly & Associates Inc, 3rd edition, 2005.

Simon P. Chung.
On the (Im)Practicality of System-Call-Based IDSs.
`http://www.cs.utexas.edu/users/phchung/publication.html`.
Accessed 09/04.

Stefan Petri.
Lastausgleich und Fehlertoleranz in Workstation-Clustern.
Shaker Verlag, May 1997.

Literature II

Niels Provos.
Systrace Interactive Policy Generation for System Calls.
http://www.citi.umich.edu/u/provos/papers/systrace-lsm/,
July 2002.
Libre Software Meeting, Bordeaux, France, accessed 09/04.

Andrew S. Tanenbaum.
Modern Operating Systems.
Prentice Hall, 2001.