AUTOMATA-BASED APPROACH FOR KERNEL TRACE ANALYSIS

Gabriel Matni

École Polytechnique de Montréal
Department of Computer Engineering
gabriel.matni@polymtl.ca

Michel Dagenais

École Polytechnique de Montréal
Department of Computer Engineering
michel.dagenais@polymtl.ca

ABSTRACT

This paper presents an automata-based approach for analyzing traces generated by the kernel of an operating system. We identified a list of typical patterns of problematic behavior, to look for in a trace, and selected an appropriate state machine language to describe them. These patterns were then fed into an off-line analyzer which efficiently and simultaneously checks for their occurrences even in traces of several gigabytes. The checker achieves a linear performance with respect to the trace size. The remaining factors impacting its performance are discussed.

I. INTRODUCTION

By carefully examining execution traces of a computer system, experts can detect problematic behavior caused by software design defects, inefficiencies as well as malicious activities. It is now possible to achieve low overhead, low disturbance tracing of multi-core Linux systems with the Linux Trace Toolkit next generation (LTTng). It provides precise, low impact, highly reentrant tracing and is used for efficiently debugging large clusters [1] as well as narrowing time constraints problems in real-time embedded applications [2]. So far, the available tools that can simplify the debugging task consist of filters and visualizers such as QNX momentics and LTTV. In this paper, we present an automata-based approach to represent patterns of problematic behavior and to automatically check for their existence in one or several large traces.

Similar work exists in the field of network based Intrusion Detection, in particular with the misuse detection systems or scenario-based systems. The State Transition Analysis Technique (STAT) [3], is used to model computer penetrations as sequences of actions that result in transitions in the security states of a system. The STAT run-time core loads and processes multiple scenario plug-ins encoded into finite state machines (FSMs), by going over the audit trails in one pass. The main features of the STAT language such as transition guards and actions are also found in the State Machine language that has an accessible, open-source compiler [4]. Similarly, Microsoft Research has developed the Abstract State Machine Language [5] to model software use cases and scenarios for validation and verification.

In the kernel tracing field, Systemtap [6] and DTrace provide a scripting language resembling C that is used to enable probe points in the kernel (instrumentation sites) and to implement their associated handlers. These handlers could be used to perform run-time checking and to generate warnings when something bad happens. The script file is translated into C code and then compiled into a binary kernel module. While this approach is interesting, it does not allow off-line analysis, and any attempt to perform complex analysis at run-time may considerably slow down the system.

We will describe in section II a set of typical patterns to be looked for in a trace, and then discuss how they were described using deterministic FSMs in section III. In section IV, the checker implementation is explained, and its performance analyzed in section V. We conclude with a brief discussion of future work.

II. PATTERNS

While the system is easily extensible at a later time, it was important to start by collecting a large representative set of patterns, in order to find a most adequate pattern description language. This representative set touches on several fields such as security, software testing and performance debugging. For sake of brevity, a representative subset is described here.

II-A. Security

The SYN flood attack is a denial of service attack that consists in flooding a server with half-open TCP connections. Signs of a SYN flood attack may be found in a kernel trace if the relevant events are instrumented. It would be very inefficient to manually look for patterns caused by such an attack, thus the interest in automating the look-up process.

Escaping a chroot jail is another attack type that can be caught on a system: a privileged process (euid=0) may want to confine its access to a subtree of the filesystem by calling the chroot() system call. If this process ever tries to open a file after the call to chroot(), then this is considered to be a security violation [7]. Indeed, a malicious user can trick the program to open the system file ../../../../../etc/shadow for example. The right way to proceed would be to call

chdir("/") right after the call to chroot(), preventing the user from ever escaping the chroot jail.

II-B. Software testing

Shared resources often require locks to be held before accessing them, to avoid race conditions. In the Linux kernel, locking is more complex than in user-space due to the different states the kernel could be in (preemption enabled, disabled, servicing an irq, etc.). Validating each and every lock acquisition has already been implemented in lockdep, the Linux kernel locks validator. For instance, it makes sure at run-time that any spinlock being acquired when interrupts are enabled has never been acquired previously in an IRQ handler. The reason is that the interrupt could happen at any time - in particular when the spinlock is already held - and when the IRQ handler tries to acquire it, the corresponding CPU will spin forever. Activating this option requires recompiling the kernel and adds a slight overhead to the system. Instead, using a kernel trace and a posteriori analysis, the same kind of validations may be performed.

Another detectable programming bug consists in accessing a file descriptor after it has been closed. This illustrates a more general class of programming errors where the usage specifications state that two particular events are logically and temporally connected.

II-C. Performance debugging

Multimedia applications, and more generally soft real-time applications, are characterized by implicit temporal constraints that must be met to provide the desired QoS [8]. Assuming that tracing the kernel scheduler has a negligible impact on the system, we can verify that temporal constraints are satisfied for one or multiple real-time applications, and whenever they are not, we can show what the system was doing at that time.

III. AUTOMATA-BASED APPROACH

Even though many existing languages are capable of expressing the different scenarios described in section II, a state-transition language was selected for the following reasons:

- Simplicity and expressiveness: the language is easy to use and provides sufficient features to express new, yet to be defined, scenarios [3] .
- Domain independent: the language may be tailored to support a wide range of patterns that relate to different fields. In the Intrusion Detection field, state-transition language is widely used to model attack signatures [3]. In model checking and Software Security, it is equally used for scenario-oriented modeling to examine security properties [7] or to verify and validate software use cases (AsmL [5]).
- Synthetic events: the state-transition approach lets us easily generate synthetic events from lower level primary events [3]. Consider for instance the SYN flood attack detection. We first model a half-open TCP connection using the state machine shown in Figure 1. When the server receives a connection request, the system moves to state S1. The server sends the acknowledgment and a timer is started. If the client sends back the acknowledgment, the system returns to state S0. Otherwise, when the timeout occurs, the system moves to S2 and a synthetic event is generated called "halfopentcp". Frequent occurrences of this synthetic event would probably mean that an attack is taking place. Synthetic events are very useful when describing even more complex patterns.

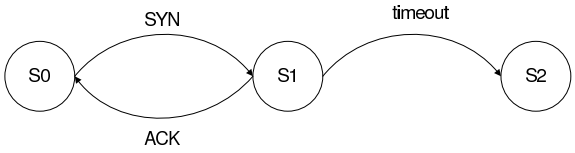

Fig. 1. Half open TCP connection

We now describe how the three following scenarios were modeled using FSMs: chroot jail escape, locking validation and real-time constraints checking.

III-A. Escaping a chroot jail

An automaton showing the sequence of system calls that may result in a security violation is shown in Figure 2. The vulnerability is explained in II-A. A call to chroot() brings the system to state S1 and saves the process id. Furthermore, a new FSM is forked in case a new chroot() call is issued by another process. The FSM fork is initiated by the transition action fork_fsm(). Any process issuing a successive call to chdir("/"), brings back the corresponding FSM to state S0, whereas a call to open() brings it to S2 and generates a warning. The machine transitions to a fourth Exit state, not shown here, and it happens whenever the exit() call is issued by the process.

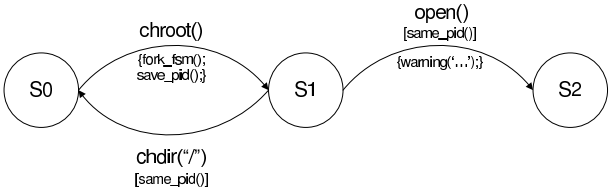

Fig. 2. Escaping the chroot jail.

III-B. Locking validation

We generate in Figure 3 an automaton that will validate a subset of the kernel locking rules. The event irq_entry() brings the system to state Irq_Handling and event irq_exit brings it back to its normal state. Any lock could be acquired either from one of the normal states (S0 or Holding_Lock) or from the Irq_Handling state. If a lock being acquired when interrupts are enabled has previously been acquired from the Irq_Handling state, the system transitions to state Potential_Deadlock. The reason is that once this lock is taken and before it gets released, if the code is interrupted by the same handler which tries to acquire the same lock, then a deadlock occurs. Similarly, if a lock previously taken when irqs were on, is now being acquired from an irq handler, then the system should also transition to the state Potential_Deadlock.

Suppose the system is in state Holding_Lock on a particular processor, a lock being held on behalf of a certain process. If this process gets scheduled out, then there is another potential deadlock due to the fact that some other process may require the same lock.

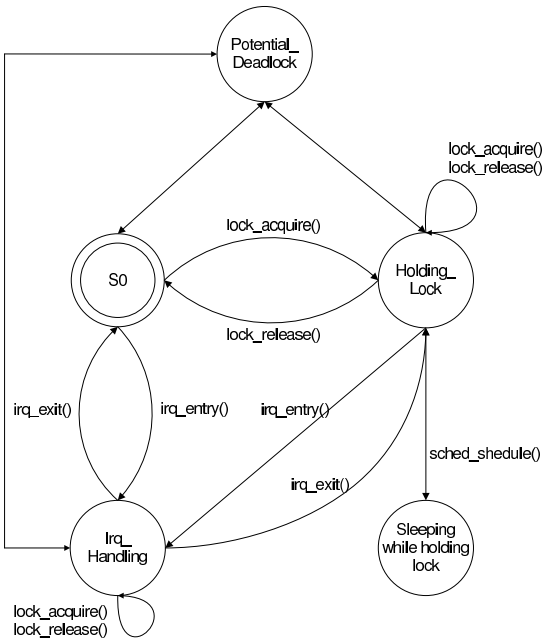

Fig. 3. Locking Validation

III-C. Real-time constraints checking

To support soft real-time applications, the kernel should respect the application's temporal constraints and therefore a predictable schedule is desired [8]. Such applications may require periodic scheduling where the period is derived from the frame rate of an audio/video stream for example. We show in Figure 4 a detailed state machine that enables us to check if the application's execution period has been respected throughout the life of the trace. Whenever it is not, we show the list of events that hindered the application's scheduling. From state Sleeping, two schedule_in() transitions bring the FSM to the Running state and save the event time stamp. The guarded transition has a higher execution priority than the other one. It computes the time difference between every two consecutive schedule_in() events. If the result is greater than a user specified threshold, a warning is generated. The event time stamp displayed by the 'warning' can then be used to determine all the preceding events once the trace is opened using the Linux Trace Toolkit Viewer (LTTV). From the Running state, the event schedule_out() brings the FSM back to the Sleeping state. The time stamp of this event is also used to compute the assigned time slice for the application, so that the transition could also trigger a warning when the time slice is less than expected.

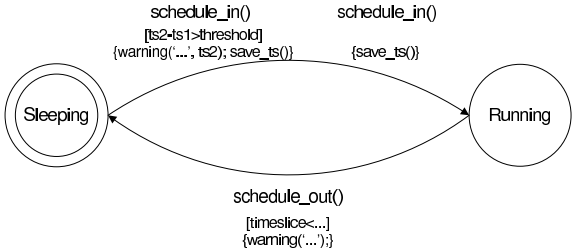

Fig. 4. Real-time constraints validation

IV. IMPLEMENTATION

In order to trace the relevant events for our analysis, we instrumented the kernel using the Linux Trace Toolkit, a low disturbance kernel and user space tracer. The information about the filesystem, inter-process communication, system-calls, memory management and kernel locking is efficiently collected, precisely time-stamped and saved at run-time. To implement the FSMs, we used the open source State Machine Compiler (SMC) that converts input files written in the state machine language into C, C++ or Java code, (other languages are supported). We show in table I a self explanatory code snippet of the language describing state S1 from Figure 2. From state S1, two transitions are possible, chdir() and open(). If the encountered event is a call to chdir, then the transition guard (between square brackets) is evaluated. In this case, if the functions same_pid() and check_new_dir() return true, then the transition is triggered and the system moves back to state S0. It is also possible to have a transition action (between braces). In our example, the call to the function warning() occurs only if the corresponding transition guard is evaluated to true.

Table I. sm language snippet

```
S1{
chdir(pid: int, newdir: char *)
    [same_pid(pid) && check_new_dir(newdir)]
    S0
open(pid: int)
    [same_pid(pid)]
    S2
    {warning(pid);destroy_fsm();}
}
```

The transition guards and actions are implemented in C and linked to the checker. The checker is a shared library that is dynamically linked to the trace reader and visualizer program called LTTV (Linux Trace Toolkit Viewer). The checker works as follows: it instantiates one FSM per pattern. For every event of interest, it registers callback functions with LTTV. When a relevant event is encountered, the checker first calls the transition(s) registered for this event and then, if needed, forks new state machine instances and adds them to its state machines list (see example in III-A). In some cases, such as the locking validation pattern, one FSM per CPU is enough. There, the checker determines on which CPU the event occurred, and only calls the transition of the FSM for that particular CPU.

Furthermore, when we instrumented the events of interest for this pattern, we noticed that the irq entry and exit events are not needed because the information could be determined from the lock_acquire() event. At this point, we simply eliminated the irq_handling state from our FSM.

We study the performance of the checker and we compare the performance of the automata-based approach with that of a dedicated implementation in section V.

V. PERFORMANCE

We used the SMC compiler to generate C code from the state machines described using the SM language. We instrumented the Linux kernel version 2.6.26 using LTTng and the tests were performed on a Pentium 4 with 512 MB of RAM. In table II we show the execution time of our checker to look up 3 different patterns: real-time constraints, file descriptors and the chroot patterns in traces of different sizes. Our results show that the execution time is linear with respect to the trace size. In fact, the performance of the checker depends on three other factors: the number of coexisting FSMs, their complexity (i.e. memory usage per state and transition) and the frequency of relevant events triggering a transition. The number of coexisting FSMs depends on the pattern in question. For instance, the fd checker has one FSM per process accessing a file descriptor whereas the real-time checker worked on just one FSM for the Movie Player (mplayer) process. We obtained similar execution times due to the fact that event sched_schedule() was occurring more frequently than events read() and write().

Table II. execution times

	500MB	1GB	1.5GB	2GB
rt checking	55s	117s	168s	252s
fd checking	57s	119s	166s	266s
chroot checking	55s	108s	166s	266s
all	67s	123s	184s	279s

The performance of the FSM checker was 4.5% slower than the dedicated version when validating the locking pattern. This was expected because it dealt with frequently occurring events leading to frequent FSM invocations. However, our approach is more generic and the overhead is often acceptable in offline analysis.

VI. CONCLUSION AND FUTURE WORK

Being able to validate a trace against a collection of predefined problematic patterns is the goal of our work. We implemented a few representative patterns using the automata-based approach and a framework that checks for their existence in kernel traces. The implemented FSM checker is slower than the dedicated version but is more generic, allowing patterns to be easily described and maintained. We would like to extend our framework to support generating synthetic events and therefore be able to simplify the task of representing large and complex scenarios.

VII. REFERENCES

[1] Martin Bligh, Mathieu Desnoyers, and Rebecca Schultz, "Linux kernel debugging on google-sized clusters," June 2007.

[2] Mathieu Desnoyers and Michel R. Dagenais, "Low disturbance embedded system tracing with linux trace toolkit next generation," in *Embedded Linux Conference 2006*, 2006.

[3] Steven T. Eckmann, Giovanni Vigna, and Richard A. Kemmerer, "Statl: An attack language for state-based intrusion detection," *Journal of Computer Security*, vol. 10, pp. 71–103, 2002.

[4] Charles W. Rapp, "http://smc.sourceforge.net/," .

[5] Mike Barnett, Wolfgang Girieskamp, and Yuri Gurevich, "Scenario-oriented modeling in asml and its intstrumentation for testing," in *Foundations of Software Engineering*, 2003.

[6] Frank Ch. Eigler, "Problem solving with systemtap," in *Ottawa Linux Symposium*, 2006.

[7] Hao Chen and David Wagner, "Mops: an infrastucture for examining security properties of software," 2002.

[8] Luca Abeni, Ashvin Goel, Charles Krasic, Jim Snow, and Jonathan Walpole, "A measurement-based analysis of the real-time performance of linux," in *Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*, 2002.