Efficient Formal Verification for the Linux Kernel

Daniel Bristot de Oliveira[1,2,3][0000−0002−4577−7855],
Tommaso Cucinotta[2][0000−0002−0362−0657], and
Rômulo Silva de Oliveira[3][0000−0002−8853−9021]

[1] RHEL Platform/Real-time Team, Red Hat, Inc., Pisa, Italy.
[2] RETIS Lab, Scuola Superiore Sant'Anna, Pisa, Italy.
[3] Department of Systems Automation, UFSC, Florianópolis, Brazil.

Abstract. Formal verification of the Linux kernel has been receiving increasing attention in recent years, with the development of many models, from memory subsystems to the synchronization primitives of the real-time kernel. The effort in developing formal verification methods is justified considering the large code-base, the complexity in synchronization required in a monolithic kernel and the support for multiple architectures, along with the usage of Linux on critical systems, from high-frequency trading to self-driven cars. Despite recent developments in the area, none of the proposed approaches are suitable and flexible enough to be applied in an efficient way to a running kernel. Aiming to fill such a gap, this paper proposes a formal verification approach for the Linux kernel, based on automata models. It presents a method to auto-generate verification code from an automaton, which can be integrated into a module and dynamically added into the kernel for efficient on-the-fly verification of the system, using in-kernel tracing features. Finally, a set of experiments demonstrate verification of three models, along with performance analysis of the impact of the verification, in terms of latency and throughput of the system, showing the efficiency of the approach.

Keywords: Verification · Linux Kernel · Automata · Testing.

1 Introduction

Real-time variants of the Linux operating system (OS) have been successfully used in many safety-critical and real-time systems belonging to a wide spectrum of applications, going from sensor networks [19], robotics [39], factory automation [17] to the control of military drones [11] and distributed high-frequency trading systems [13, 10], just to mention a few. However, for a wider adoption of Linux in next-generation cyber-physical systems, like self-driving cars [42], automatic testing and formal verification of the code base is increasingly becoming a non-negotiatable requirement. One of the areas where it is mostly difficult and non-trivial to adopt such techniques is the one of the kernel, due to its inherent complexity. This need has fomented the development of many formal models for

the Linux kernel, like the Memory Model [2] and formal verification of spinlock primitives [28]. However, Linux lacks a methodology for runtime verification that can be applied broadly throughout all of the in-kernel subsystems.

Some complex subsystems of Linux have been recently modeled and verified by using automata. For example, modeling the synchronization of threads in the PREEMPT_RT Linux kernel achieved practical results in terms of problems spotted within the kernel [33] (and fixes being proposed afterwards). As a consequence, the kernel community provided positive feedback, underlining that the *event* and *state* abstractions used in automata look natural to the modeling of the kernel behavior, because developers are already accustomed to using and interpreting event traces in these terms [31, 30].

The problem, however, is that the previously proposed approach [33] relies on tracing events into an in-kernel buffer, then moving the data to user-space where it is saved to disk, for later post-processing. Although functional, when it comes to tracing high-frequency events, the act of in-kernel recording, copying to user-space, saving to disk and post-processing the data related to kernel events profoundly influences the timing behavior of the system. For instance, tracing scheduling and synchronization-related events can generate as many as 900000 events per second, and more than 100 MB per second of data, per CPU, making the approach non-practical, especially for big muti-core platforms.

An alternative could be hard-coding the verification in the Linux kernel code. This alternative, however, is prone not to become widely adopted in the kernel. It would require a considerable effort for acceptance of the code on many subsystems. Mainly because complex models can easily have thousands of states. A second alternative would be maintaining the verification code as an external *patchset*, requiring the users to recompile the kernel before doing the checking, what would inhibit the full utilization of the method as well. An efficient verification method for Linux should unify the flexibility of using the dynamic tracing features of the kernel while being able to perform the verification with low overheads.

Paper Contributions. This paper proposes an efficient automata-based verification method for the Linux kernel, capable of verifying the correct sequences of in-kernel events as happening at runtime, against a theoretical automata-based model that has been previously created. The method starts from an automata-based model, as produced through the well-known Supremica modeling tool, then it auto-generates `C` code with the ability of efficient transition look-up time in *O(1)* for each hit event. The generated code embedding the automaton is compiled as a module, loaded *on-the-fly* into the kernel and dynamically associated with kernel tracing events. This enables the run-time verification of the observed in-kernel events, compared to the sequences allowed by the model, with any mismatch being readily identified and reported. The verification is carried out in kernel space way more efficiently than it was possible to do in user-space, because there is no need to store and export the whole trace of occurred events. Indeed, results from performance analysis of a kernel under verification show

that the overhead of the verification of kernel operations is very limited, and even lower than merely activating tracing for all of the events of interest.

2 Background

This section provides the background for the two main concepts used for the verification of Linux: the automata-based formal method used for modeling, and the tracing mechanism within the kernel at the basis of the verification process.

2.1 Automata and Discrete Event System

A *Discrete Event System* (DES) can be described in various ways, for example using a *language* (that represents the valid sequences of events that can be observed during the evolution of the system). Informally speaking, an automaton is a formalization used to model a set of well-defined rules that define such a language.

The evolution of a DES is described with all possible sequence of events $e_1, e_2, e_3, ...e_n$, $e_i \in E$, defining the language $\mathcal{L}$ that describes the system.

There are many possible ways to describe the language of a system. For example, it is possible to use regular expressions. For complex systems, more flexible modeling formats, like automaton, were developed.

Automata are characterized by the typical directed graph or state transition diagram representation. For example, consider the event set $E = \{a, b, g\}$ and the state transition diagram in Figure 1, where nodes represent system states, labeled arcs represent transitions between states, the arrow points to the initial state and the nodes with double circles are *marked states*, i.e., safe states of the system.

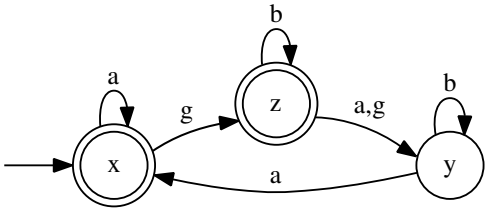

Fig. 1. State transitions diagram (based on Fig. 2.1 from [7]).

Formally, a deterministic automaton, denoted by G, is a tuple

$$G = \{X, E, f, x_0, X_m\} \tag{1}$$

where:

- X is the set of states
- E is the finite set of events
- $f : X \times E \to X$ is the transition function. It defines the state transition in the occurrence of a event from E in the state X.
- x_0 is the initial state
- $X_m \subseteq X$ is the set of marked states

For instance, the automaton G shown in Figure 1 can be defined as follows:

- $X = \{x, y, z\}$
- $E = \{a, b, g\}$
- $f : (x, a) = x; (y, a) = x; (z, b) = z; (x, g) = z; (y, b) = y; (z, a) = (z, g) = y.$
- $x_0 = x$
- $X_m = \{x, z\}$

The automaton starts from the initial state x_0 and moves to a new state $f(x_0, e)$ upon the occurrence of an event e. This process continues based on the transitions for which f is defined.

Informally, following the graph of Figure 1 it is possible to see that the occurrence of event a, followed by event g and a will lead from the initial state to state y. The language $\mathcal{L}(G)$ generated by an automaton $G = \{X, E, f, x_0, X_m\}$ consists of all possible chains of events generated by the state transition diagram starting from the initial state.

Given a set of *marked states*, i.e., possible final or safe states when modeling a system, an important language generated by an automaton is the *marked language*. This consists of the set of words in $\mathcal{L}(G)$ that lead to marked states, and it is also called the language *recognized* by the automaton.

Automata theory also enables operations among automata. An important operation is the *parallel composition* of two or more automata that are combined to compose a single, augmented-state, automaton. This allows for merging two or more automata models into one single model, constituting the standard way of building a model of an entire system from models of its individual components [7].

2.2 Linux tracing

Linux has an advanced set of tracing methods, which are mainly applied in the runtime analysis of kernel latencies and performance issues [27]. The most popular tracing methods are the `function tracer` that enables the trace of kernel functions [38], and the `tracepoint` that enables the tracing of hundreds of events in the system, like the *wakeup* of a new thread or the occurrence of an interrupt. But there are many other methods, like `kprobes` that enable the creation of `dynamic tracepoints` in arbitrary places in the kernel code [22], and composed arrangements like using the `function tracer` and `tracepoints` to examine the code path from the time a task is woken up to when it is scheduled.

An essential characteristic of the Linux tracing feature is its efficiency. Nowadays, almost all Linux based operating systems (OSes) have these tracing methods enabled and ready to be used in production kernels. Indeed, these methods

```
sh-2038  [002] d... 16230.043339: ttwu_do_wakeup <-try_to_wake_up
sh-2038  [002] d... 16230.043339: check_preempt_curr <-ttwu_do_wakeup
sh-2038  [002] d... 16230.043340: resched_curr <-check_preempt_curr
sh-2038  [002] d... 16230.043343: sched_wakeup: comm=cat pid=2040 prio=120 target_cpu=003
```

Fig. 2. `Ftrace` Output.

have nearly zero overhead when disabled, thanks to the extensive usage of runtime code modification techniques, that allow for a greater efficiency than using conditional jumps when tracing is disabled. For instance, when the `function tracer` is disabled, a `no-operation` assembly instruction is placed right at the beginning of all traceable functions. When the `function tracer` is enabled, the `no-operation` instruction is overwritten with an instruction that calls a function that will *trace* the execution, for instance by appending information into an in-kernel trace buffer. This is done at runtime, without any need for a reboot. A `tracepoint` works similarly, but using a jump label [14]. The mentioned tracing methods are implemented in such a way that it is possible to specify how an event will be handled dynamically, at runtime. For example, when enabling a `tracepoint`, the function responsible to handle the event is specified through a proper in-kernel API.

Currently, there are two main interfaces by which these features can be accessed from user-space: `perf` and `Ftrace`. Both tools can hook to the trace methods, processing the events in many different ways. The most common action is to record the occurrence of events into a trace-buffer for post-processing or human interpretation of the events. Figure 2 shows the output of the `Ftrace` tracing functions and `tracepoints`. The recording of events is optimized by the usage of *per-cpu* lock-less trace buffers. Furthermore, it is possible to take actions based on events. For example, it is possible to record a *stacktrace*.

These tracing methods can also be leveraged for other purposes. Similarly to `perf` and `Ftrace`, other tools can also hook a function to a tracing method, non-necessarily for the purpose of providing a trace of the system execution to the user-space. For example, the Live Patching feature of Linux uses the `function tracer` to hook and deviate the execution of a problematic function to a revised version of the function that fixes a problem [36].

3 Related work

This section overviews major prior works related to the technique introduced in this paper, focusing on automata-based modeling of various Linux kernel subsystems, use of formal methods for other OS kernels, and finally the use of other formal methods to assess the correctness of various Linux kernel functions.

3.1 Automata-based Linux modelling

A number of works exist making use of automata-based models to verify correctness of Linux kernel code. The work presented in [29] uses trace and automata

to verify conditions in the kernel. The paper presents models for SYN-flood, escaping from a chroot jail, validation of locking and of real-time constraints. The LTTng tracer [41] is used to compare the models to the kernel execution. The models are very simple and presented as proof of concept. There are only five states in the largest model, which is related to locking validation. There are only two states in the real-time constraints model. Despite its simplicity, this paper corroborates the idea of connecting automata to tracing as a layer of translation from kernel to formal methods, including aspects of Linux real-time features.

State-aware/Stateful robustness testing [26] is an important area that uses formal system definition. Robust testing is also used in the OS context as a fault tolerance technique [37]. A case study of state-based robustness testing is presented in [15] that includes the OS states of a real-time version of Linux. The results show that the OS state plays a significant role in testing for corner cases that are not covered by traditional robustness verification. Another relevant project for Linux is SABRINE [16], an approach using tracing and automata for *state-aware robustness testing* of OSes. SABRINE works as follows: It traces the interactions among components of the OS in the first step. The software then extracts state models from the traces automatically. The traces are processed in this phase in order to find sequences of similar functions, to be grouped, forming a pattern. Later, similar patterns are grouped into clusters. The last step is the generation of the behavioral model from the clusters. A behavioral model consists of event-connected states in the finite-state automata (FSA) format.

The ability to extract models from the operating system depends on the operating system components specification and their interfaces. The paper targets not a system component, but the set of mechanisms used to synchronize NMI, IRQ, and thread operations. The analyzed events are present in most subsystems, such as disabling interruptions and preemption, or locks.

SABRINE was later improved by the TIMEOUT approach [40] which records the time spent in each state. The FSA is then created using timed automata. The worst-case execution time observed during the profiling phase is used as the Timed-FSA's timing parameter, so timing errors can also be detected.

3.2 Formal methods and OS kernels

Verification of an operating system kernel, with its various components, is a particularly challenging area.

Some works that addressed this issue include the BLAST tool [21], where control flow automata were used, combining existing state-space reduction techniques based on verification and counter-example-driven refinement with *lazy abstraction*. This enables on-demand refinement of specification parts by selecting more specific predicates to add to the model while the model checker is running, without the need to revisit parts of the state space that are not affected by the refinements. Interestingly, for the Linux and Microsoft Windows NT kernels, authors applied the technique to verify the security properties of OS drivers. The technique required instrumentation of the original drivers, inserting

a conditional jump to an error handler, and a model of the surrounding kernel behavior to enable the verification that the faulty code could ever be reached.

The SLAM [4] static code analyzer shares major goals with BLAST, enabling C programs to be analyzed to detect violations of certain conditions. SLAM is also used within the Static Driver Verifier (SDV) framework [3] to check Microsoft Windows device drivers against a set of rules. For example, it has been used to detect improper use of the Windows XP kernel API in some drivers. SATABS [5] and CBMC [24] are verification tools used within the DDVerify [43] framework to check synchronization constructs, interrupts and deferred tasks.

MAGIC [8] is a tool for automatic verification of sequential C programs that uses finite state machine specifications. The tool can analyze a direct acyclic graph of C functions by extracting a finite state model from the source code and then reducing the verification to a problem of boolean satisfiability (SAT). The verification is performed by checking the specification against an increasingly refined sequence of abstractions until either it is verified or a counter-example is found. This allows the technique to be used with relatively large models, along with its modular approach, avoiding the need to enumerate the state-space of the entire system. MAGIC was used to verify the correctness of a number of functions involved in system calls handling mutexes, sockets and packet handling in the Linux kernel. The tool was also later extended to handle concurrent software systems [9], although authors focused on verifying correctness and liveness in presence of message-passing based concurrency without variable sharing. Authors were able to find a bug in the source code of Micro-C/OS, although the bug had already been fixed in a new release when they notified the developers.

Other remarkable works have also been carried out evaluating the formal correctness of a whole micro-kernel, such as seL4 [23], regarding the adherence of the compiled code to its expected behavior stated in formal terms. seL4 also includes precise worst-case execution time analysis [6]. These findings were possible thanks to the simplicity of the seL4 micro-kernel, e.g. semi-preemptability.

3.3 Formal methods and the Linux kernel community

The adoption of formal methods is not new to the Linux kernel community, especially in the kernel development and debugging workflow.

Indeed, the `lockdep` mechanism [12] built into the Linux kernel is a remarkable work in this area. By observing the order of execution and the calling context of lock calls, Lockdep is able to identify errors in the use of locking primitives that could eventually lead to deadlocks. The mechanism includes detecting errors in the acquisition order of multiple (nested) locks across multiple kernel code paths, and detecting common errors in handling spinlocks across the IRQ handler vs process context, such as acquiring a spinlock from the process context with enabled IRQs as well as from an IRQ handler. By applying the technique based on locking classes instead of individual locks, the number of different lock states that the kernel must keep is reduced.

A formal memory model is introduced in [2] to automate the verification of the consistency of core kernel synchronization operations, across a wide range of

supported architectures and associated memory consistency models. The Linux memory model ended up being part of the official Linux release, adding the Linux Kernel Memory Consistency Model (LKMM) subsystem, an array of tools that formally describe the Linux memory consistency model, and also producing "litmus tests" in the form of kernel code that can be executed and tested directly.

The TLA+ formalism [25] has also been successfully applied to discover bugs in the Linux kernel. Examples of problems discovered or confirmed by using TLA+ include the correctness of memory management locking during a context switch and fairness properties of the arm64 ticket spinlock implementation [28].

These recent results created interest in the potential of using formal methods in Linux development. Therefore, the present paper describes our proposed technique for validation at *runtime* of allowed kernel events sequences, as specified through an automata-based model. As highlighted above, the technique fills an empty spot in the related literature, focusing on *efficient* verification that is achieved by: 1) tracking relevant kernel events at a proper abstraction level, leveraging the `perf` and `Ftrace` subsystems, but 2) without any need to actually collect a full trace of the relevant events from the kernel to user-space for further analysis: events sequences are directly checked inside the kernel leveraging efficient code automatically generated from the automata-based model, characterized by a *O(1)* event processing time adding very small overheads, even lower than those arising merely for *tracing* the relevant events. This will be shown through experimental results in Section 5.

4 Efficient Formal Verification for the Linux Kernel

An overarching view of the approach being proposed in this paper is displayed in Figure 3. It has three major phases. First, the behavior of a part of the Linux kernel is modeled using automata, using the set of events that are available in the tracing infrastructure[4]. The model is represented using the *.dot* `Graphviz` format [20]. The *.dot* format is open and widely used to represent finite-state machines and automata. For example, the Supremica modeling tool [1] supports exporting automata models using this format.

Figure 4 presents the example of an automaton for the verification of in-kernel scheduling-related events. The model specifies that the event *sched_waking* cannot take place while preemption is enabled, in order not to cause concurrency issues with the scheduler code (see [33] for more details).

In the second step, the `.dot` file is translated into a `C` data structure, using the `dot2c` tool [5]. The auto-generated code follows a naming convention that allows it to be linked with a kernel module skeleton that is already able to refer to the generated data structures, performing the verification of occurring events

[4] These can be obtained for example by running: `sudo cat /sys/kernel/debug/tracing/available_events`.

[5] The tools, the verification modules, the BUG report, high-resolution figures and FAQ are available in the companion page [32].

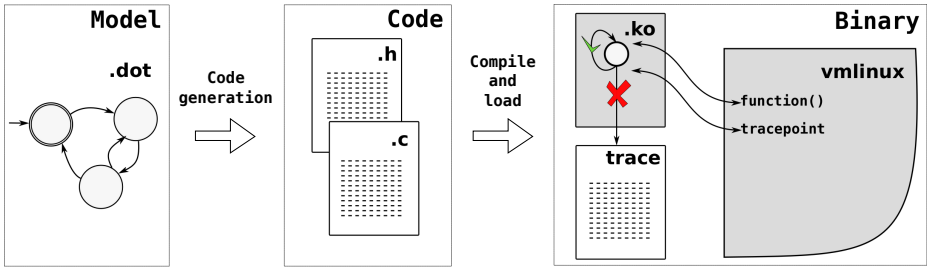

Fig. 3. Verification approach.

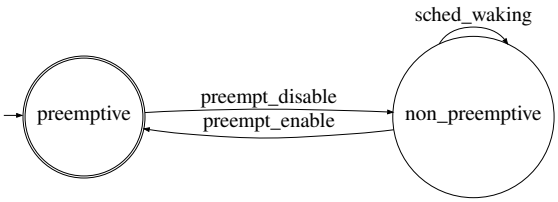

Fig. 4. Wake-up In Preemptive (*WIP*) Model.

in the kernel, according to the specified model. For example, the automaton in Figure 4 is transformed into the code in Figure 5.

The `enum states` and `events` provide useful identifiers for states and events. As the name suggests, the `struct automaton` contains the automaton structure definition. Its corresponding `C` version contains the same elements of the formal definition. The most critical element of the structure is `function`, a matrix indexed in constant time *O(1)* by `curr_state` and `event` (as shown in the `get_next_state()` function in Figure 6). Likewise, for debugging and reporting reasons, it is also possible to translate the event and state indexes into strings in constant time, using the `state_names` and `event_names` vectors.

Regarding scalability, although the matrix is not the most efficient solution with respect to the memory footprint, in practice, the values are reasonable for nowadays common computing platforms. For instance, the Linux Task Model Automata presented in [33], with 9017 states and 20103 transitions, resulted in a binary object of less than 800KB, a reasonable value even for nowadays Linux-based embedded systems. The automaton structure is static, so no element changes are allowed during the verification. This simplifies greatly the needed synchronization for accessing it. The only information that changes is the variable that saves the *current state* of the automata, so it can easily be handled with atomic operations, that can be a single variable for a model that represents the entire system. For instance, the model in Figure 4 represents the state of a CPU (because the preemption enabling status is a *per-cpu* status variable in Linux), so there is a *current state* variable *per-cpu*, with the cost of (*1 Byte * the number of CPUs of the system*). The simplicity of automaton defini-

```
enum states {
  preemptive = 0,
  non_preemptive,
  state_max
};

enum events {
  preempt_disable = 0,
  preempt_enable,
  sched_waking,
  event_max
};

struct automaton {
  char *state_names[state_max];
  char *event_names[event_max];
  char function[state_max][event_max];
  char initial_state;
  char final_states[state_max];
};

struct automaton aut = {
  .event_names = { "preempt_disable", "preempt_enable",
                   "sched_waking" },
  .state_names = { "preemptive", "non_preemptive" },
  .function = {
    { non_preemptive,          -1,             -1 },
    {              -1, preemptive, non_preemptive },
  },
  .initial_state = preemptive,
  .final_states = { 1, 0 }
};
```

Fig. 5. Auto-generated code from the automaton in Figure 4.

```
char get_next_state(struct automaton *aut, enum states curr_state,
                    enum events event) {
  return aut->function[curr_state][event];
}
```

Fig. 6. Helper functions to get the next state.

tion is a crucial factor for this method: all verification functions are *O(1)*, the definition itself does not change during the verification and the sole information that changes has a minimal footprint.

In the last step, the auto-generated code from the automata, along with a set of helper functions that associate each automata event to a kernel event, are compiled into a kernel module (a `.ko` file). The model in Figure 4 uses only `tracepoints`. The *preempt_disable* and *preempt_enable* automaton events are connected to the `preemptirq:preempt_disable` and `preemptirq:preempt_enable` kernel events, respectively, while the *sched_waking* automaton event is connected to the `sched:sched_waking` kernel event. The Sleeping While in Atomic (*SWA*) model in Figure 7 also uses tracepoints for *preempt_disable* and *enable*, as well as for *local_irq_disable* and *enable*. But the *SWA* model also uses `function tracers`.

One common source of problems in the PREEMPT_RT Linux is the execution of functions that might put the process to sleep, while in a non-preemptive code section [34]. The event *might_sleep_function* represents these functions. At initialization time, the *SWA* module *hooks* to a set of functions that are known to eventually putting the thread to sleep.

Note that another noteworthy characteristic of the proposed framework is that, by using user-space probes [18], it is also possible to perform an integrated automata-based verification of both user and kernel-space events, without requiring code modifications.

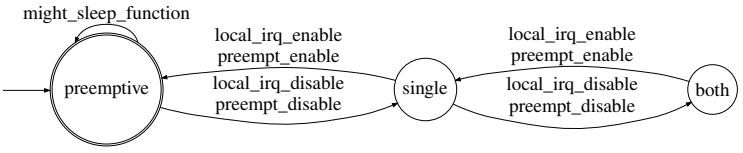

Fig. 7. Sleeping While in Atomic (*SWA*) model.

The kernel module produced as just described can be loaded at any time during the kernel execution. During initialization, the module connects the functions that handle the automaton events to the kernel tracing events, and the verification can start. The verification keeps going on until it is explicitly disabled at runtime by unloading the module.

The verification output can be observed via the tracing file regularly produced by Ftrace. As performance is a major concern for runtime verification, debug messages can be disabled of course. In this case, the verification will produce output only in case of problems.

An example of output is shown in Figure 8. In this example, in Line 1 a *debug* message is printed, notifying the occurrence of the event *preempt_enable*, moving the automaton from the state *non_preemptive* to *preemptive*. In Line 2, *sched_waking* is not expected in the state *preemptive*, causing the output of the *stack trace*, to report the code path in which the problem was observed.

The problem reported in Figure 8 is the output of a *real bug* found in the kernel while developing this approach. The bug was reported to the Linux kernel mailing list, including the verification module as the test-case for reproducing the problem [5].

5 Performance evaluation

Being efficient is a key factor for a broader adoption of a verification method. Indeed, an efficient method has the potential to increase its usage among Linux developers and practitioners, mainly during development, when the vast majority of complex testing takes place. Therefore, this section focuses on the performance

```
bash-1157  [003] ....2..    191.199172: process_event: non_preemptive -> preempt_enable = preemptive safe!
bash-1157  [003] dN..5..    191.199182: process_event: event sched_waking not expected in the state preemptive
bash-1157  [003] dN..5..    191.199186: <stack trace>
=> process_event
=> __handle_event
=> ttwu_do_wakeup
=> try_to_wake_up
=> irq_exit
=> smp_apic_timer_interrupt
=> apic_timer_interrupt
=> rcu_irq_exit_irqson
=> trace_preempt_on
=> preempt_count_sub
=> _raw_spin_unlock_irqrestore
=> __down_write_common
=> anon_vma_clone
=> anon_vma_fork
=> copy_process.part.42
=> _do_fork
=> do_syscall_64
=> entry_SYSCALL_64_after_hwframe
```

Fig. 8. Example of output from the proposed verification module, as occurring when a problem is found.

of the proposed technique, by presenting evaluation results on a real platform verifying models, in terms of the two most important performance metrics for Linux kernel (and user-space) developers: *throughput* and *latency*.

The measurements were conducted on an HP ProLiant BL460c G7 server, with two six-cores Intel Xeon L5640 processors and 12GB of RAM, running a Fedora 30 Linux distribution. The kernel selected for the experiments is the Linux PREEMPT_RT version *5.0.7-rt5*. The real-time kernel is more sensible for synchronization as the modeled preemption and IRQ-related operations occur more frequently than in the mainline kernel.

5.1 Throughput evaluation

Throughput evaluation was made using the *Phoronix Test Suite* benchmark [35], and its output is shown in Figure 9. The same experiments were repeated in three different configurations. First, the benchmark was run in the system *as-is*, without any tracing nor verification running. Then, it was run in the system after enabling verification of the *SWA* model. Finally, a run was made with the system being traced, only limited to the events used in the verified automaton. It is worth mentioning that tracing in the experiments means only recording the events. The complete verification in user-space would still require the copy of data to user-space and the verification itself, which would add further overhead.

On the CPU bound tests (Crypto, CPU Stress and Memory Copying), both trace and verification have a low impact on the system performance. In contrast, the benchmarks that run mostly on kernel code highlights the overheads of both methods. In all cases, the verification performs better than tracing. The reason

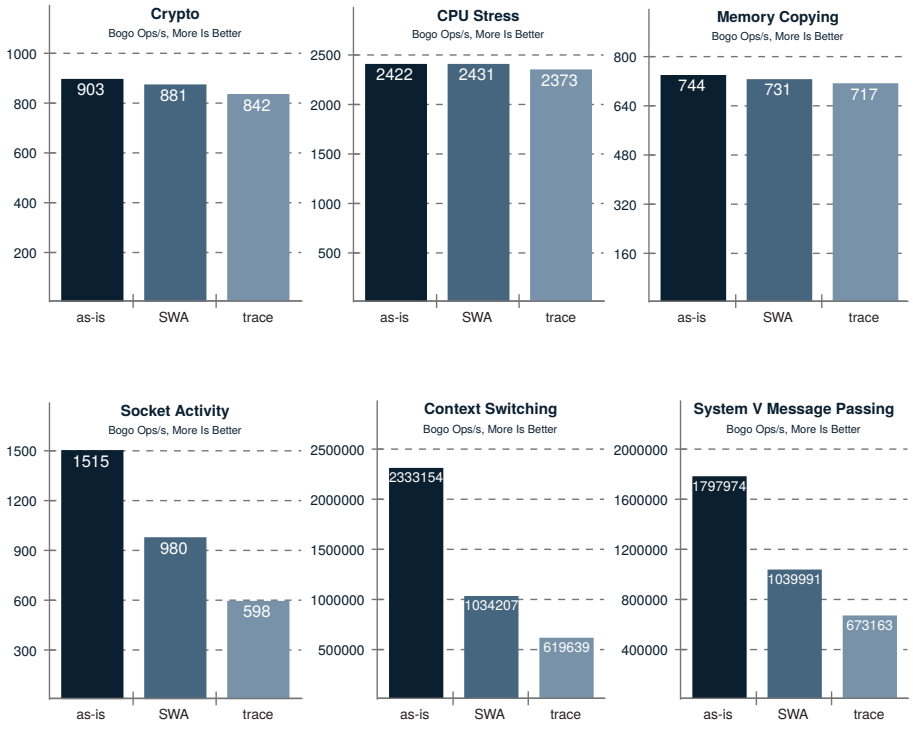

Fig. 9. Phoronix Stress-NG Benchmark Results: *as-is* is the system without tracing nor verification; *SWA* is the system while verifying *Sleeping While in Atomic* automata in Figure 11 and with the code in Figure 5; and the *trace* is the system while tracing the same events used in the *SWA* verification.

is that, despite the efficiency of tracing, the amount of data that has to be manipulated costs more than the simple operations required to do the verification, essentially the cost of looking up the next state in memory in *O(1)*, and storing the next state with a single memory write operation.

5.2 Latency evaluation

Latency is the main metric used when working with the PREEMPT_RT kernel. The latency of interest is defined as the delay the highest real-time priority thread suffers from, during a new activation, due to in-kernel synchronization. Linux practitioners use the `cyclictest` tool to measure this latency, along with `rteval` as background workload, generating intensive kernel activation.

Two models were used in the latency experiment. Similarly to Section 5.1, the *SWA* model was evaluated against the kernel *as-is*, and the kernel simply

tracing the same set of events. In addition, the *Need Re-Schedule* (*NRS*) model in Figure 10 was evaluated. It describes the synchronization events that influence the latency, and it is part of the model previously described in [33][6]. The *NRS* measurements were made on the same system but configured as a single CPU[7].

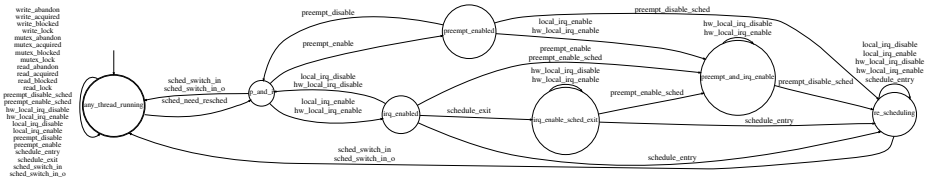

Fig. 10. Need Re-Sched forces Scheduling (*NRS* model) from [33] [5].

Consistently with the results obtained in the throughput experiments, the proposed verification mechanism is more efficient than the sole tracing of the same events. This has the effect that the `cyclictest` latency obtained under the proposed method, shown in Figure 11 (*SWA*/*NRS* curves), is more similar to the one of the kernel *as-is* than what is obtained while just tracing the events.

6 Conclusions and Future work

The increasing complexity of the Linux kernel code-base, along with its increasing usage in safety-critical and real-time systems, pushed towards a stronger need for applying formal verification techniques to various kernel subsystems. Nonetheless, two factors have been placing a barrier in this regard: 1) The need of complex setups, even including modifications and re-compilation of the kernel; 2) The excessively poor performance exhibited by the kernel while under tracing, for collecting data needed in the verification, typically carried out in user-space.

The solution for both problems seemed to be controversial: the usage of in-kernel tracing along with user-space post-processing reduces the complexity of the setup, but incurs the problem of having to collect, transfer to user-space and process large amounts of data. On the other hand, the inclusion of verification code "hard-coded" in the kernel requires more complex setups, with the need for applying custom patches and recompiling the kernel, with said patches being quite cumbersome to maintain as the kernel evolves over time.

This paper tackles these two problems by using the standard tracing infrastructure available in the Linux kernel to dynamically attach verification code to a non-modified running kernel, by exploiting the mechanism of dynamically loadable kernel modules. Furthermore, the verification code is semi-automatically

[6] Note that supporting the full model in [33] is not yet possible with the tool being presented in this paper, due to additional changes needed within the kernel. Therefore, this is still work in progress.

[7] This is a restriction from [33].

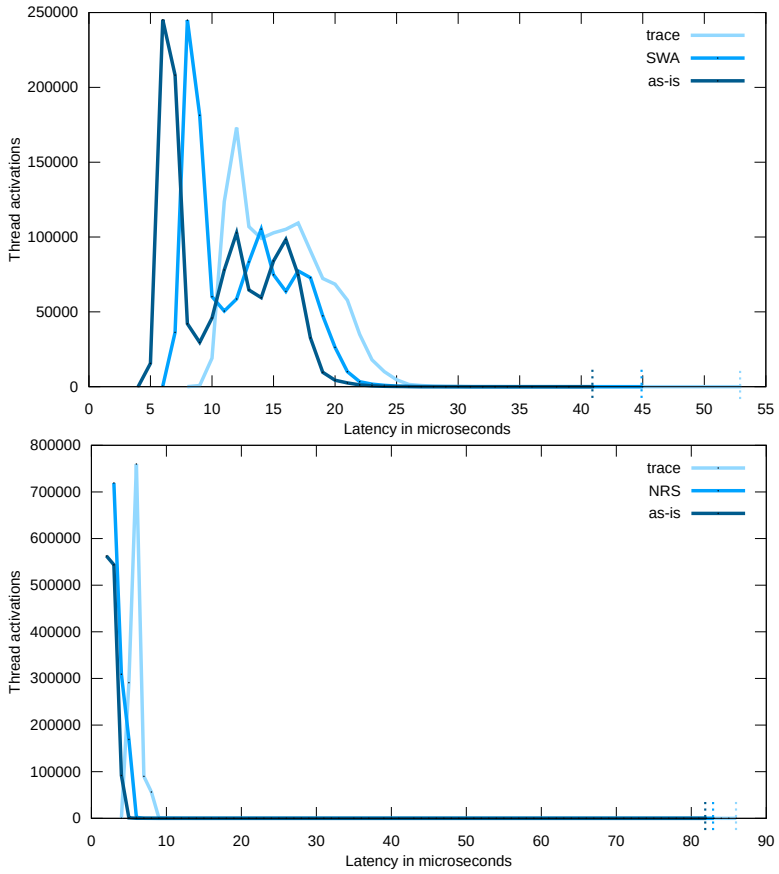

Fig. 11. Latency evaluation, using the *SWA* model (top) and the *NRS* model (bottom).

generated from standard automata description files, as can be produced with open editors. The presented benchmark results show that the proposed technique overcomes standard tracing and user-space processing of kernel events to be verified in terms of performance. Moreover, the proposed technique is more efficient than merely tracking the events of interest just using tracing features available in the kernel.

Regarding possible future work on the topic, the usage of parametric and timed-automata would open the possibility of using more complex and complete verification methods, not only addressing the logical and functional behavior, but also dealing with the timing behavior. In terms of efficiency of the implementation, a hot-topic in the Linux kernel tracing community is the in-kernel processing of data via *eBPF*, as established already with in-kernel packet processing. This might be a worthwhile avenue to explore and compare with the current method of using a dynamically loadable module, in which part of the code has been auto-generated.

References

1. Akesson, K., Fabian, M., Flordal, H., Malik, R.: Supremica - an integrated environment for verification, synthesis and simulation of discrete event systems. In: 2006 8th International Workshop on Discrete Event Systems. pp. 384–385 (July 2006). https://doi.org/10.1109/WODES.2006.382401
2. Alglave, J., Maranget, L., McKenney, P.E., Parri, A., Stern, A.: Frightening Small Children and Disconcerting Grown-ups: Concurrency in the Linux Kernel. In: Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 405–418. ASPLOS '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3173162.3177156, http://doi.acm.org/10.1145/3173162.3177156
3. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: Technical Report MSR-TR-2004-08 – SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft – Microsoft Research. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2004-08.pdf (January 2004)
4. Ball, T., Rajamani, S.K.: The SLAM Project: Debugging System Software via Static Analysis. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 1–3. POPL '02, ACM, New York, NY, USA (2002). https://doi.org/10.1145/503272.503274
5. Basler, G., Donaldson, A., Kaiser, A., Kroening, D., Tautschnig, M., Wahl, T.: Satabs: A bit-precise verifier for c programs. In: Flanagan, C., König, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 552–555. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
6. Blackham, B., Shi, Y., Chattopadhyay, S., Roychoudhury, A., Heiser, G.: Timing analysis of a protected operating system kernel. In: Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS11). pp. 339–348. Vienna, Austria (November 2011)
7. Cassandras, C.G., Lafortune, S.: Introduction to Discrete Event Systems. Springer Publishing Company, Incorporated, 2nd edn. (2010)
8. Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. IEEE Transactions on Software Engineering **30**(6), 388–402 (June 2004). https://doi.org/10.1109/TSE.2004.22
9. Chaki, S., Clarke, E., Ouaknine, J., Sharygina, N., Sinha, N.: Concurrent software verification with states, events, and deadlocks. Formal Aspects of Computing **17**(4), 461–483 (Dec 2005). https://doi.org/10.1007/s00165-005-0071-z, https://doi.org/10.1007/s00165-005-0071-z
10. Chishiro, H.: Rt-seed: Real-time middleware for semi-fixed-priority scheduling. In: 2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC)
11. Condliffe, J.: U.s. military drones are going to start running on linux. https://gizmodo.com/u-s-military-drones-are-going-to-start-running-on-linu-1572853572 (Jul 2014)
12. Corbet, J.: The kernel lock validator. https://lwn.net/Articles/185666/ (May 2006)
13. Corbet, J.: Linux at NASDAQ OMX. https://lwn.net/Articles/411064/ (Oct 2010)
14. Corbet, J.: Jump label. https://lwn.net/Articles/412072/ (October 2010)
15. Cotroneo, D., Di Leo, D., Natella, R., Pietrantuono, R.: A case study on state-based robustness testing of an operating system for the avionic domain. In: Flammini,

F., Bologna, S., Vittorini, V. (eds.) Computer Safety, Reliability, and Security. pp. 213–227. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
16. Cotroneo, D., Leo, D.D., Fucci, F., Natella, R.: Sabrine: State-based robustness testing of operating systems. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering. pp. 125–135. ASE'13, IEEE Press, Piscataway, NJ, USA (2013). https://doi.org/10.1109/ASE.2013.6693073, https://doi.org/10.1109/ASE.2013.6693073
17. Cucinotta, T., Mancina, A., Anastasi, G.F., Lipari, G., Mangeruca, L., Checcozzo, R., Rusina, F.: A real-time service-oriented architecture for industrial automation. IEEE Transactions on Industrial Informatics **5**(3), 267–277 (Aug 2009). https://doi.org/10.1109/TII.2009.2027013
18. Dronamraju, S.: Linux kernel documentation - uprobe-tracer: Uprobe-based event tracing. https://www.kernel.org/doc/Documentation/trace/uprobetracer.txt (May 2019)
19. Dubey, A., Karsai, G., Abdelwahed, S.: Compensating for timing jitter in computing systems with general-purpose operating systems. In: 2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing. pp. 55–62 (March 2009). https://doi.org/10.1109/ISORC.2009.28
20. Ellson, J., Gansner, E., Koutsofios, L., North, S.C., Woodhull, G.: Graphviz – open source graph drawing tools. In: International Symposium on Graph Drawing. pp. 483–484. Springer (2001)
21. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 58–70. POPL '02, ACM, New York, NY, USA (2002). https://doi.org/10.1145/503272.503279
22. Hiramatsu, M.: Linux tracing technologies: Kprobe-based event tracing. https://www.kernel.org/doc/html/latest/trace/kprobetrace.html (May 2019)
23. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal Verification of an OS Kernel. In: Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles. pp. 207–220. SOSP '09, ACM, New York, NY, USA (2009). https://doi.org/10.1145/1629575.1629596
24. Kroening, D., Tautschnig, M.: Cbmc – c bounded model checker. In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 389–391. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
25. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. **16**(3), 872–923 (May 1994). https://doi.org/10.1145/177492.177726
26. Lei, B., Liu, Z., Morisset, C., Li, X.: State based robustness testing for components. Electron. Notes Theor. Comput. Sci. **260**, 173–188 (Jan 2010). https://doi.org/10.1016/j.entcs.2009.12.037, http://dx.doi.org/10.1016/j.entcs.2009.12.037
27. Linux Kernel Documentation: Linux tracing technologies. https://www.kernel.org/doc/html/latest/trace/index.html (May 2019)
28. Marinas, C.: Formal methods for kernel hackers. URL: https://linuxplumbersconf.org/event/2/contributions/60/attachments/18/42/FormalMethodsPlumbers2018.pdf (2018)
29. Matni, G., Dagenais, M.: Automata-based approach for kernel trace analysis. In: 2009 Canadian Conference on Electrical and Computer Engineering. pp. 970–973 (May 2009). https://doi.org/10.1109/CCECE.2009.5090273

30. de Oliveira, D.B.: How can we catch problems that can break the preempt_rt preemption model? https://linuxplumbersconf.org/event/2/contributions/190/ (Nov 2018)
31. de Oliveira, D.B.: Mind the gap between real-time linux and real-time theory. https://www.linuxplumbersconf.org/event/2/contributions/75/ (Nov 2018)
32. de Oliveira, D.B.: Companion page: Efficient formal verification for the linux kernel. http://bristot.me/efficient-formal-verification-for-the-linux-kernel/ (May 2019)
33. de Oliveira, D.B., Cucinotta, T., de Oliveira, R.S.: Untangling the Intricacies of Thread Synchronization in the PREEMPT_RT Linux Kernel. In: Proceedings of the IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC). Valencia, Spain (May 2019)
34. de Oliveira, D.B., de Oliveira, R.S.: Timing analysis of the PREEMPT_RT Linux kernel. Softw., Pract. Exper. **46**(6), 789–819 (2016). https://doi.org/10.1002/spe.2333
35. Phoronix Test Suite: Open-source, automated benchmarking. www.phoronix-test-suite.com (May 2019)
36. Poimboeuf, J.: Introducing kpatch: Dynamic kernel patching. https://www.redhat.com/en/blog/introducing-kpatch-dynamic-kernel-patching (February 2014)
37. Pullum, L.L.: Software Fault Tolerance Techniques and Implementation. Artech House, Inc., Norwood, MA, USA (2001)
38. Rostedt, S.: Secrets of the Ftrace function tracer. Linux Weekly News (January 2010), available at: http://lwn.net/Articles/370423/ [last accessed 09 May 2017]
39. San Vicente Gutiérrez, C., Usategui San Juan, L., Zamalloa Ugarte, I., Mayoral Vilches, V.: Real-time linux communications: an evaluation of the linux communication stack for real-time robotic applications (Aug 2018), https://arxiv.org/pdf/1808.10821.pdf
40. Shahpasand, R., Sedaghat, Y., Paydar, S.: Improving the stateful robustness testing of embedded real-time operating systems. In: 2016 6th International Conference on Computer and Knowledge Engineering (ICCKE). pp. 159–164 (Oct 2016). https://doi.org/10.1109/ICCKE.2016.7802133
41. Spear, A., Levy, M., Desnoyers, M.: Using tracing to solve the multicore system debug problem. Computer **45**(12), 60–64 (Dec 2012). https://doi.org/10.1109/MC.2012.191
42. The Linux Foundation: Automotive grade linux. https://www.automotivelinux.org/ (May 2019)
43. Witkowski, T., Blanc, N., Kroening, D., Weissenbacher, G.: Model Checking Concurrent Linux Device Drivers. In: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering. pp. 501–504. ASE '07, ACM, New York, NY, USA (2007). https://doi.org/10.1145/1321631.1321719