Runtime Verification in the Linux Kernel: Monitoring Kernel Memory Allocation

Andre Lee (ael226)

December 2020

***Abstract*- Runtime verification (RV) is a tool with great potential in software engineering, able to monitor program executions against formal properties and detect violations. However, despite great progress in research in implementing practical RV artifacts in new environments, little work has been done in the realm of operating systems, even less in ones popularly used in the industry. In this paper new artifacts are presented to tackle this issue; we seek to present a monitoring setup for memory properties[1] in the Linux kernel, monitoring programs not specific to language or those that must implement certain packages.**

1 Introduction

Runtime verification is an integral component of monitor-oriented programming. The general structure of runtime verification is to *instrument*, or retrieve running events from, a software system. A chronologically-ordered list of events, a *trace*, is then passed to a *monitor*, which can then detect problems in the program execution. Detectable problems are those that violate *specifications*, or formulae denoting software properties important to developers.

Detecting problems in programs this way has the potential to be very useful for different stages in software development. Coupled with test programs, this can become bug detection; runtime verification can even be implemented as a feature in final products, able to supervise software systems and fix any issues that it detects.

Decades of research in the RV community have gone into improving instrumentation, streamlining monitoring, creating specifications, etc. However, very few research has been done in the realm of monitoring one of the most widespread computing environments: operating systems. In recent years, papers on operating systems applications of RV have included these papers [2],[3],[4] , and each contain some drawbacks in their utility to general software development:

- Reinbacher, Függer, and Brauer [2] implemented an RV framework on the Field Programmable Gate Array platform, as research into RV into embedded systems. This is interesting in expanding environments for RV, but the specificity of the system makes it difficult for developers to use or adopt.
- Efremov and Shchepetkov [3] implemented offline monitoring of security policies for a specific Linux distro, Astra Linux, a distro almost exclusively used by the Russian government. Monitoring security is important, but offline monitoring leaves out the possibility of tracking in real-time. Also, similar to the above example, the specificity of the implementation makes it hard to adopt.
- The work of Huang, Erdogan et al. [4] in a more popular operating system, ROS, makes runtime verification more practical in monitoring real-time message passing and even user commands. However applications are generally limited to robotics.

In this paper, we extend their goals to include a more popular environment. We demonstrate monitoring implemented in a widely-used Linux distro (Ubuntu 16.04 LTS). Memory safety is universally important in any computing environment concerned about safety, and is especially important in programs relying on C[1], such as the Linux kernel. The monitoring tools we present here are designed to monitor memory properties for two ubiquitous events in kernel C: `kmalloc` and `kfree`. In the future we hope the designs presented are useful in generalizing RV to more practical developer environments. A GitHub repository is provided for those who wish to demo our artifacts.

1.1 Background

Here we define terms used in this document, beyond the general definitions given earlier:

- *Online and offline monitoring*: Offline monitoring in RV is when events of a program execution are recorded by the instrumentation; a monitor is run on these events afterward. Online monitoring allows for real-time RV, processing each event as it occurs.

- *kmalloc and kfree*: `kmalloc` in kernel C is the equivalent to `malloc` as it allocates memory, while `kfree` de-allocates the memory. However, `kmalloc` allocates memory that is both virtually and physically contiguous, and as such is the source of several vulnerabilities in the Linux kernel.

- *Hypervisor*: A hypervisor in this context is a level of abstraction sitting between the 'supervisor' (operating system) and the computing hardware. It is able to read events from the operating system and send signals to its processes.

- *Kernel module*: A kernel module contains code that can be loaded to extend the kernel functionality at runtime. It can serve as a medium between kernel and user space in the Linux kernel; for example, most drivers are implemented as kernel modules.

1.2 Monitoring Kernel Memory

In Roşu's paper[1] on monitoring memory safety in kernel C, there are certain proofs that are important in the discussion of what safety properties we hope to monitor, and how we seek to monitor them. Given the semantics of kernel C, a kernel C program is memory safe if none of its possible executions gets stuck in a non-final state. Following from this, Roşu demonstrates memory safety is in general undecidable for program executions (as they might not halt).

But even for finite program executions memory safety is undecidable. This comes from the non-deterministic nature of memory allocation: `kmalloc` takes as input a block size and returns a pointer to memory of that size in the kernel space; this return value can point to any arbitrary point in available memory, and once de-allocated can be reallocated. This means determining memory safety for a program under all possible combinations of memory locations is not decidable.

Roşu proposes a variation to kernel, dubbed SafeKernelC, that uses symbolic representations of memory, abstracting away the non-determinism of reallocation and memory collision. He proves that using the semantics of SafeKernelC, deciding memory safety for finite program executions is possible. Our implementation focuses on monitoring kernel memory accesses by `kmalloc` and `kfree` in finite program traces. The specifications we use in our monitors explore this extra restriction and a possible relaxation.

2 Implementation

2.1 Specifications

2.1.1 Specification 1: All allocated memory must be freed, and memory can only be freed if it has already been allocated.

This specification might seem long-winded, but represented by a regular expression, it is: for a given pointer parameter y, we have expression (`kmalloc`(y) `kfree`(y))*. Unfreed memory in a machine with finite memory (as real computers tend to be), can lead to lower computational resources over time. Freeing memory multiple times, or even without allocating memory, can lead to undefined behavior. It might free up memory used up by another process, which might then be re-allocated for a different use, causing errors as unexpected sets of data will be written and read from that same memory location. As a finite state machine (FSM), it is presented in Figure 1:

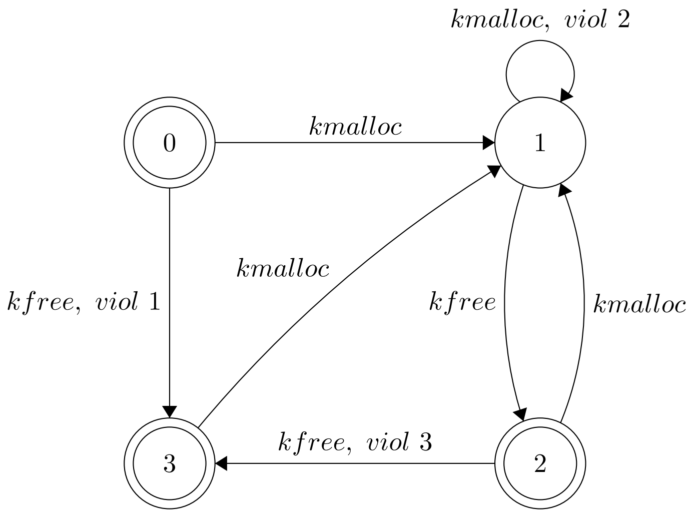

Figure 1: Finite-State Machine for Specification 1

This is the FSM our monitor implementation will be using, mapping violations to transitions as they occur; when the trace ends, non-terminating states will also produce violations:

1. Memory freed without being allocated.
2. Memory re-allocated without being freed.
3. Memory freed without being re-allocated.
4. (After trace ends): Memory allocated but not freed.

2.1.2 Specification 2: Memory cannot be reallocated once it has been freed.

This specification may seem harsh, but we are only enforcing this one process at a time. This approximates the restriction of symbolic memory used in SafeKernelC for a running process. As a regular expression, for a given memory pointer y, we have expression (`kmalloc`(y) `kfree`(y)). The FSM for this specification is much simpler, producing only one violation: Memory location reallocated after freed.

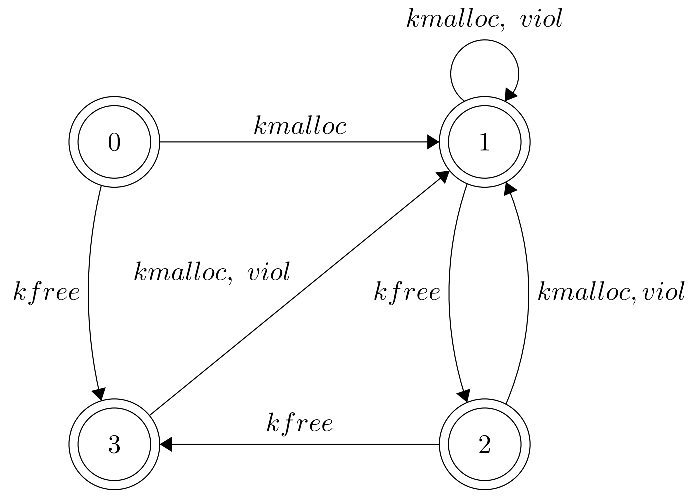

Figure 2: Finite-State Machine for Specification 2

2.1.3 Specification 3: Memory cannot be *immediately* re-allocated once it has been freed.

This is a relaxation of the symbolic memory representation. Within a process, once a pointer y has been freed, y cannot be the next pointer allocated by `kmalloc`. If a different pointer x is given with an allocation, then y is able to be re-allocated. This relaxation has been chosen as symbolic memory representation is more unrealistic, but it is still an approximate measure of an issue in software design if it is detected that the same pointer is allocated, freed, and re-allocated in rapid succession.

To enforce this property, monitors for multiple parameters must be able to communicate. As such, our implementation uses separate FSM for each parameter, but there is a stack they share to facilitate "communication". You may call this a multi-parameter pushdown automaton.

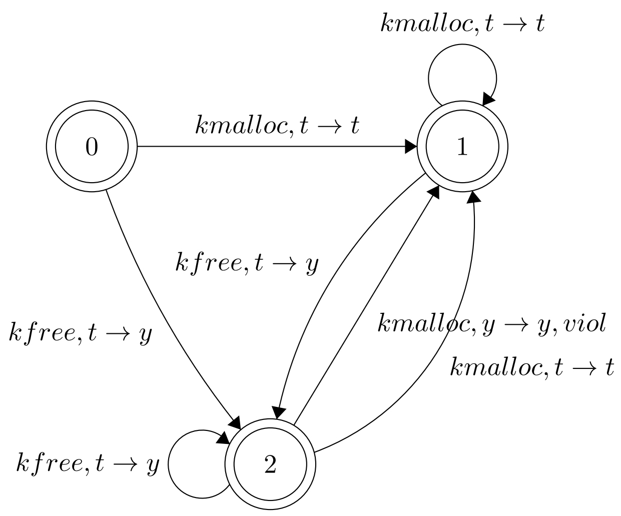

Figure 3: Finite-State Gadget for Specification 3 Machine

For any parameter y, when `kfree`(y) is called, the top of the stack is pushed off and y is pushed on. When memory is re-allocated after a pointer is freed, if y is returned from `kmalloc` and y is on the top of the stack, a violation is returned. This is what is meant by "immediate".

2.2 Instrumentation

2.2.1 Design Decision: Kernel Module vs. Hypervisor

For collecting function calls from the Linux kernel in our instrumentation, there were two approaches considered: either a hypervisor to block and execute code on specific syscalls, or a kernel module able to probe the kernel at specific debug symbols. We decided on using kernel modules due to feasability issues: the hypervisor would need to not only simulate the operating system we are monitoring (e.g. VirtualBox), but also be able to collect signals and respond to them. We believe that ad-hoc implementing such a tool would have to break important safety properties to access execution levels above the operating system, so we opt for using kernel modules. However, kernel modules have their own limitations as we will discuss later.

2.2.2 SystemTap

For instrumentation of the Linux kernel, one of the tools used by Efremov and Shchepetkov [3] was SystemTap, an open-source project designed to probe kernel and user space functions in Linux systems. We decided to use SystemTap for the flexibility of putting probes to gather arguments and return values, as well as the power of executing arbitrary kernel code whenever a probe is hit. The SystemTap tool takes in a script specifying these things, and runs a kernel module to probe and perform the operations. It is even able to probe when a specific process ends, allowing the use of a special "end" event to communicate the end of a trace. However, there were some issues encountered during the process of developing monitoring tools:

- **SystemTap is buggy on some distros**: SystemTap was originally designed to work with Red Hat and Fedora distros; it has been extended to Debian, GNOME, Ubuntu, etc. but on some versions it has bugs that need to be patched.

- **SystemTap instruments itself**: As an example, if a probe was set to the `printf` function, and the code executed at that probe used `printf`, it would generate an infinite loop that locks up the machine. The issue is that unlike a hypervisor, kernel modules operate at the same execution level as the rest of the kernel, so they access the same libraries and functions as the rest of the system. As such, our instrumentation is unable to use kernel memory allocation functions if we wish to instrument them.

- **SystemTap has tight memory/time constraints**: Modules generated with SystemTap do not have the ability to store a lot of events within their allocated memory. Additionally, to keep the system running smoothly, it is unable to block that long at probe points to execute code (if it detects a wait, it will begin to drop events).

2.3 Monitoring

2.3.1 Monitoring Algorithm

Our implementation includes both offline and online monitoring. Both types use a similar simple monitoring algorithm, written in C. Each specification is implemented in a monitoring program, with each program handling state transitions for all parameters. Given an object `[function, parameter]` from the instrumentation's probes, the monitor parses the two parts separately, looking up the state corresponding with the `parameter`. Using a `switch{case:}` statement to simulate a FSM, the monitor obtains a new state in O(1) time and assigns the new state to the `parameter`.

Due to SystemTap's instrumenting itself, these monitor algorithms are designed to run without being able to allocate more memory. This means they are initialized with all the memory the Linux system allows them to have, and no more. This means all data structures for the monitors are implemented with arrays of limited size. This is one of the critical constraints on our current monitor implementations.

2.3.2 Offline Monitoring

For offline monitoring, in order to stop the kernel from instrumenting itself, the SystemTap script was designed to store a buffer of all events in the trace, and upon the end of the target process would write the trace onto a file to be read by monitors. The monitors used for offline monitoring used arrays to store parameters and states. They would use a simple linear search to find the correct state, since overhead is not a pertinent issue for offline monitoring. The runtime complexity of the offline monitors is O(np), where n is the number of events in the trace, and p the number of parameters. Since p is upwards bounded by n, it can be simplified to O(n^2).

2.3.3 First Approach to Online Monitoring

For online monitoring, traditional RV would have a monitor run alongside the target process, with the instrumentation facilitating blocking in the target process and communicating with the monitor.

In our first implementation, the monitor runs in user space while the instrumentation ran in kernel space. To facilitate communication, we used another kernel module with ioctl calls that used locks to ensure FIFO delivery of all events. Every time the instrumentation detected an event of interest, it would send it to the monitor, and then block until it receives a confirmation from the monitor. The monitor would process the event, send a confirmation, and then block until the next event arrived.

The approach failed because of one of the issues with SystemTap: the time constraints made it drop incoming events whenever it would block. We needed an approach that would not need extensive blocking, a way to condense monitoring of each event into a single quick function call.

2.3.4 Second Approach to Online Monitoring

Our second approach was to implement the monitor for each specification as a kernel module, not a running process. Rather, they are kernel objects that store FSM information for every parameter, where monitoring is done by sending events to a monitoring API to update this state.

Due to greater concerns about overhead, the monitors' storage of parameters utilize a binary search-tree for O($log(p)$) lookup times. Whenever the instrumentation detects an event of interest, it calls into the API for the monitors, passing in the event information. Therefore, the full runtime complexity of this second approach is O($n\ log(n)$).

2.4 Initial System Validation

To validate the ability of SystemTap to capture all events of interest, and to measure the correctness of monitors, all were run on a toy target program. Using a kernel module with ioctl calls to call `kmalloc` and `kfree`, the toy program is a user space program that allows us to control which kernel functions were called and in what order. All monitors (offline and online) were run on the toy program, and the traces and violations checked against the program.

An interesting thing to note, is that whenever a process is started in Linux, SystemTap detects several `kfree` calls before the program code is run. This indicates that the Linux kernel garbage-collects previously allocated memory when a process starts, perhaps in anticipation of memory usage by that process. And when monitoring Specification 1, this means violations of type 1 will appear when starting any process.

The toy program was some validation that the instrumentation is able to capture all events of interest, and that the monitors accurately reflected the specifications. However, it is a short program, and the experimentation was needed to stress-test the current implementation.

3 Evaluation

3.1 Research Questions

1. How much overhead does running our monitors incur on longer traces and stress tests?

2. With our constraints due to SystemTap, do our monitors or instrumentation end up dropping events or parameters on longer test programs?

3. Comparing Specification 2 and Specification 3, how do the number of violations compare when monitoring real projects, and what does this say about the target programs?

3.2 Experiment Description

For the experiments, we are running our monitoring implementations on the DaCapo benchmark [5]. The reason we chose the DaCapo testing suite was twofold:

1. The DaCapo benchmark is a well-known set of tests used in a wide variety of research in the RV community, and in future work can continue to be used to measure against this current project.

2. The benchmark uses Java which is separate from the monitoring implementation in C. While we are monitoring kernel C events, we want to use this experiment to demonstrate monitoring of kernel memory regardless of the programming language used in the target program.

We are running this test on Ubuntu Linux 16.04 LTS, using SystemTap version 4.0 and DaCapo version 9.12; in case of unforeseen behavior, we are simulating the Ubuntu OS in a VirtualBox environment. Our Java environment was OpenJDK 8. Each program in the version 9.12 DaCapo benchmark will be run three times without any instrumentation or monitoring, three times with online monitoring, and three times with offline monitoring. We will measure runtime overhead for our monitoring techniques based on average runtimes, and measure violations for each specification in online and offline monitor outputs.

Table 1: Experimental Results of Monitoring on the DaCapo Benchmark

		Online Monitoring					Offline Monitoring				
program	time(ms)	time(ms)	OH(%)	Spec 1	Spec 2	Spec 3	time(ms)	OH(%)	Spec 1	Spec 2	Spec 3
avrora	4522	4632	2.43	1516	713	37	5687	25.76	894	497	37
batik	FAIL	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
eclipse	58154	57545	-1.05	7861	3779	579	72923	25.4	5221	4278	71
fop	2752	2829	2.8	758	846	25	2854	3.71	479	377	16
h2	6280	6121	-2.53	939	67	21	6243	-0.6	939	65	21
jython	10157	10816	6.49	4028	1931	59	12927	27.27	3550	1832	58
luindex	1981	2090	5.5	4654	5281	1340	2119	6.97	797	448	56
lusearch-fix	3978	3776	-5.08	819	186	17	5822	46.35	93	18	0
lusearch	3883	4257	9.63	959	202	16	5058	30.26	2058	1856	263
pmd	4150	4311	3.88	2661	1522	78	4402	6.07	478	368	0
sunflow	7740	7718	-0.28	366	75	24	8551	10.48	977	791	9
tomcat	FAIL	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
tradebeans	9660	8828	-8.61	11965	9657	1212	9406	-2.63	289	293	69
tradesoap	33623	32604	-3.03	8943	7540	1204	45044	33.97	10742	7193	1250
xalan	6495	6298	-3.03	370	87	22	8773	35.07	3450	2121	134
Avg			0.55					19.08			

3.3 Results

The results of the experiment are recorded in Table 1. First off, we must mention that the DaCapo benchmarks of `batik` and `tomcat` failed on our system, with neither monitoring nor instrumentation, so results from those tests were discarded. The overhead measurements (OH) are recorded simply by taking the percentage difference between DaCapo runtime without monitoring and runtime with monitoring. For each specification monitored, the average number of violations found are recorded. Going into more detail, many of the programs had high variance in the amount of violations incurred (much of the memory management is abstracted through the JDK, and the non-deterministic nature of memory allocation comes into play). There was also some variance in runtime, and we will discuss more in the next section.

3.3.1 Measured Overhead

At first, it might seem counter-intuitive that offline monitoring overhead (that takes into account only instrumentation runtime) is on average much higher than online monitoring overhead (that contains both instrumentation and monitoring runtime). However, offline monitoring also contains file I/O operations, which are very time-costly. This makes the overhead increase even further with the longest traces, with long runtimes to write every event into a file. This leads to higher overhead, averaging over 20%.

On the other hand, due to the high variance in runtimes for online monitoring, the very low overhead in online monitoring is likely inaccurate. We can see that online monitoring overhead correlates with program runtime less than offline monitoring does, so that is not a good predictor, but since overhead never went above 10% during the experiment, a more realistic estimate of overhead would be about 5-10%.

3.3.2 Losing Probes and Events

During the short validation testing, both online and offline monitoring detected the same events and produced identical violations. However, in stress testing on the DaCapo benchmark, while the reported violations are very similar in some projects, for the majority of DaCapo tests the reported violations diverged quite wildly. By examining the individual trials closer, online monitoring tended to have much lower variance in reported violations compared to offline. Output produced by offline monitoring also sometimes contained errors due to unrecognizable strings or hitting EOF. Since online monitoring generally reports more violations when the average diverges from that of offline monitoring, we can reason that the instrumentation for offline monitoring is not recording all events.

Originally, time constraints and self-instrumentation made SystemTap need to store all events in a buffer before writing the entire buffer to a file at once as the target process is ending. This becomes an issue as limited memory means that there is an upper bound on the number of events that can be monitored offline. This is one possible source of offline monitoring losing events.

The second is from file I/O in the context of the experiment. Because multiple trials are run, we added little time between finishing instrumentation and running offline monitors. As such, offline monitoring might be missing events due to read-after-write errors.

3.3.3 Violations from Specifications 2 and 3

From the data, as expected all monitors report less violations of Specification 3 than 2, about 60-90% less. Looking at individual trials, a reduction of over 95% can happen regularly. While thousands of violations of Spec. 2 may be reported, usually <100 violations of Spec. 3 are reported.

Looking at DaCapo, these benchmarks are large test programs with high memory usage. We expect memory to constantly be re-allocated, especially within the context of a small virtual machine. However, since DaCapo is a longstanding benchmark, consisting of well-tested programs developed over many years, in most circumstances in designing software, there would be no need to consecutively free and re-allocate the same memory address so many times.

But there were some projects that had many Spec. 3 violations. Looking at `tradesoap`, with over a thousand violations with both online and offline monitoring, the test program involves populating and re-populating a database, then running many transactions on it. With large amounts of memory that needs to be contiguous, old memory would need to be recycled at high rates as the program runs.

With the non-deterministic memory allocation, violating these specifications of kernel C might not always directly indicate bugs, but large amounts of violations might call for a closer look at software design and implementation.

4 Discussion

In this project we have implemented artifacts capable of online and offline monitoring of three specifications for monitoring kernel memory allocation. Using SystemTap instrumentation, both monitoring styles have been validated and are consistent on small programs with less events.

Testing on the popular DaCapo benchmark demonstrates that the challenges posed by integrating SystemTap with our monitors remain great. While overhead appears to be generally quite low, true measures of accuracy for reported violations could not be found, but we are able to infer some issues with offline monitoring due to measured discrepancies with online monitoring.

Our artifacts, including monitors, SystemTap instrumentation and test scripts, are available on GitHub at `https://github.com/andreezlee/OS-Runtime-Verification`. We plan to share these approaches with others and gather more data on additional benchmarks.

5 Future Work

Future work includes improvements on integrating SystemTap with kernel modules to remove the restrictions on timing and memory that have revealed further problems during our system evaluation. Creating a custom version of SystemTap that allows probing of memory accesses and not just function calls would be useful; `use-after-free` violations were an initial idea but proved at this current point unfeasible to instrument.

In addition, in order to make operating system RV more practical for software engineers, we can add more features to current monitors. Besides adding more specifications that can more directly reveal bugs, when violations are detected, monitors might print stack traces to find how kernel functions are used in production-level code and the sources of the violations.

There is also another way to use monitors: responding to kernel violations and fixing issues in real-time. For example, changing `kmalloc` return values might be necessary if it is deemed better to have a process end due to NULL pointer than to allow that program access to kernel memory. These are all extensions of our artifacts, and we hope they will lead to the multi-purpose monitoring of different types of kernel properties in RV.

6 Self-Assessment

1. In trying to implement RV in a new area, I have learned a lot about working with new specifications, implementing monitors, and the importance of instrumentation. While most of the papers we read this semester focused on monitors and specs, most of my time on this project was spent on the instrumentation: getting it to set probes properly, connect to monitors, and attempting to overcome its limitations. I believe the project has given me valuable experience and technical insights in working with software engineering and its testing/correctness.

2. I would definitely have tested SystemTap more thoroughly to discover its restrictions or explored alternatives more deeply. Another thing I would have done differently is to have started by examining kernel modules more closely, because a lot of time was spent on crafting modules that would go between monitoring and instrumentation.

3. Due to my topic being a little out of range of typical RV, I am unsure if the professor could have helped me more on the project. I am not sure if he could have connected me to anyone who was exploring a similar topic or any relevant resources.

4. I believe I tried to accomplish what I set out to do. Despite the hiccups, I believe my vision for the project has been mostly implemented, with some remaining issues. I would give myself a 3.5/5.

7 References

[1] Roşu G., Schulte W., Şerbănuţă T.F. (2009) Runtime Verification of C Memory Safety. In: Bensalem S., Peled D.A. (eds) Runtime Verification. RV 2009. Lecture Notes in Computer Science, vol 5779. Springer, Berlin, Heidelberg. `https://doi.org/10.1007/978-3-642-04694-0_10`

[2] Reinbacher T, Függer M, Brauer J. Runtime verification of embedded real-time systems. Form Methods Syst Des. 2014;44:203-239. `doi:10.1007/s10703-013-0199-z`

[3] Efremov D., Shchepetkov I. (2020) Runtime Verification of Linux Kernel Security Module. In: Sekerinski E. et al. (eds) Formal Methods. FM 2019 International Workshops. FM 2019. Lecture Notes in Computer Science, vol 12233. Springer, Cham. `https://doi.org/10.1007/978-3-030-54997-8_12`

[4] Huang J. et al. (2014) ROSRV: Runtime Verification for Robots. In: Bonakdarpour B., Smolka S.A. (eds) Runtime Verification. RV 2014. Lecture Notes in Computer Science, vol 8734. Springer, Cham. `https://doi.org/10.1007/978-3-319-11164-3_20`

[5] Blackburn, Stephen M. et al. 2006. The DaCapo benchmarks: java benchmarking development and analysis. In Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA '06). Association for Computing Machinery, New York, NY, USA, 169–190. DOI:`https://doi.org/10.1145/1167473.1167488`