System Calls Instrumentation for Intrusion Detection in Embedded Mixed-Criticality Systems

Marine Kadar
SYSGO GmbH, Germany
marine.kadar@sysgo.com

Sergey Tverdyshev
SYSGO GmbH, Germany
sergey.tverdyshev@sysgo.com

Gerhard Fohler
Kaiserslautern University, Germany
fohler@eit.uni-kl.de

Abstract

System call relative information such as occurrences, type, parameters, and return values are well established metrics to reveal intrusions in a system software. Many Host Intrusion Detection Systems (HIDS) from research and industry analyze these data for continuous system monitoring at runtime. Despite a significant false alarm rate, this type of defense offers high detection precision for both known and zero-day attacks. Recent research focuses on HIDS deployment for desktop computers. Yet, the integration of such run-time monitoring solution in mixed-criticality embedded systems has not been discussed. Because of the cohabitation of potentially vulnerable non-critical software with critical software, securing mixed-criticality systems is a non trivial but essential issue. Thus, we propose a methodology to evaluate the impact of deploying system call instrumentation in such context. We analyze the impact in a concrete use-case with PikeOS real-time hypervisor.

2012 ACM Subject Classification Security and privacy → Embedded systems security; Security and privacy → Intrusion detection systems

Keywords and phrases Instrumentation, Mixed-criticality, Real-Time, System Calls, Host Intrusion Detection Systems

Digital Object Identifier 10.4230/OASIcs.CERTS.2019.2

Funding This work has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 764785, FORA—Fog Computing for Robotics and Industrial Automation.

Acknowledgements The authors would like to thank internal reviewers at SYSGO for their valuable feedback.

1 Introduction

Mixed-criticality systems consolidate software applications of different criticality levels on a single hardware platform. For example in a car, an Android operating system which manages infotainment can run together on a single hardware, with an AUTOSAR adaptive runtime environment that controls lights management. In such system, security is a major and non-trivial issue at design- and run-time. On one hand, software components with a low criticality (e.g., Android) offer full management and handling of critical resources (e.g., network, console I/O, file-system access) to their non-critical application workloads. On the other hand, highly critical software components running in parallel depend on the availability and correct operations of the same resources or shared hardware.

The intrusion detection approach assesses that the system environment potentially contains flaws which could be exploited by an attacker. To identify suspicious activity, this

© Marine Kadar and Sergey Tverdyshev and Gerhard Fohler;
licensed under Creative Commons License CC-BY

4th International Workshop on Security and Dependability of Critical Embedded Real-Time Systems (CERTS 2019).
Editors: Mikael Asplund and Michael Paulitsch; Article No. 2; pp. 2:1–2:13

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

technique continuously observes the system at runtime. Suspicious activity corresponds either to an anomaly, defined as a deviation from normal activity, or to an intrusion, which is identified using the signature of previously known intrusions. Intrusion detection presents a strong potential as it can contribute to detect zero-day attacks. Intrusion Detection Systems (IDS) are notably used for securing IT infrastructures (e.g. anti-virus software, network monitoring, host behavior analytics). They also represent an important segment of research. In Host Intrusion Detection Systems (HIDS), system call patterns appear to be a good metric to identify security threats. System call monitoring has indeed been investigated since late nineties, starting with [20]. Recent research uses these data mainly for machine-learning based HIDS.

To the best of our knowledge, existing machine-learning based HIDS from the industry and research focus on deployment for generic computers. The use of such defense in a mixed-criticality system has yet not been investigated. Protecting embedded mixed-criticality systems is however essential, as these systems are exposed to same threats than non-critical systems. Adapting HIDS in this context is very challenging. One one hand, because it is deployed at system level, HIDS can potentially compromise system execution; e.g. a real-time application missing its deadline. On the other hand, HIDS can degrade time performance of non critical workloads. Thus, we need to ensure that the add-on does not alter system execution and constraints.

To implement system call monitoring, we have to integrate system call tracing in the mixed-criticality OS kernel. The new feature should be fast and predictable; its impact on real-time system execution needs to be determined. We propose an evaluation of the feasibility of system call instrumentation for deployment in a mixed-criticality system. After introducing our approach to evaluate the impact, we apply it in a concrete example, using the real-time hypervisor PikeOS [8].

The remainder of the paper is organized as follows. In section 2, we summarize related work. Section 3 introduces the methodology. Section 4 describes the experiment, while section 5 presents the results. We discuss in section 6 the security impact of intrusion detection as well as further implementation of the detection, We also mention other interesting data to detect intrusions as well as their potential impact on system's real-time constraints. Finally, section 7 concludes this paper.

2 Related Work

2.1 Security by Design

One main approach to enforce security, as well as safety, in mixed-criticality systems is to enforce the whole system's design, by providing strong isolation for all workloads. The OS kernel must be as protected in term of security and safety, as the most critical workload. To address this need, Multiple Independent Levels of Security (MILS) [2] architecture is a concept based on separation kernel and partitioning to ensure data and program separation. Nevertheless, software isolation can not be enough to ensure system isolation. The misuse of shared hardware resources by an attacker can potentially lead to system exploit, such as side channels attacks. For example, Spectre [12] and Meltdown [15] attacks succeed in leaking the memory of complete programs, by leveraging specific CPU vulnerability. As explained in [10], isolation of hardware resources is a non-trivial problem; because of system complexity, we generally can not assess 100% isolation. Hence, we need further protection to ensure system security.

2.2 Control Flow Integrity

Control Flow Integrity is a type of intrusion detection solution based on continuous monitoring. It observes programs executed in the system at run-time to detect any deviation from a defined normal execution policy. The policy can be defined for each individual program at compilation time, using specific tools such as LLVM compiler [18]. It generally provides a list of allowed branch transfers in the program address space.

TrackOS [17] is an example of real-time OS (RTOS), which integrates a CFI monitor. The RTOS hosts different workloads called tasks. One main task with higher execution privilege, the CFI Monitor, monitors all the other ones at run-time. Motivated by certification reasons, the monitored programs remain unchanged. A graph of execution is generated with static analysis from each untrusted program executable. Time credit is allocated by the user for each task, including the CFI Monitor. During its execution time slot, the monitor checks the stack of monitored workloads using the statically defined policy. Hence, the security impact on execution time for monitored workloads is controlled by the user.

Because of a higher frequency of control flow transfers over system call instructions, The impact of CFI on performance represents a major limitation to its deployment. For this reason, we do not consider CFI approach for our research.

2.3 System Call Tracing for Security

Host Intrusion Detection System (HIDS) is a well developed security solution, which relies on system call collection at runtime. It has been well investigated for two decades, since [20] highlighted a statistical correlation of intrusion patterns with certain system calls sequences. In literature, many papers focus on HIDS performance in terms of detection precision and false positive rates. It is an efficient way to detect intrusions; detection precision is generally above 90%, despite a high false positive rate, which can easily reach 15% in recent research. For example [9] compares several analysis models with average 90% detection precision and 15% false positives rates. [13] develops and HIDS based on Markov chains with 90% detection precision and 20% false positives rate. False postives can be reduced by modifying the model's parameters. Nevertheless, this usually induces a loss of precision: [5] compares six models on the same dataset, five of which show a lower false positive rate below 5%, while the detection rate varies between 40% and 100%.

Unfortunately, very few publications focusing on HIDS describe or analyze the impact on time execution for integrating such software at system run-time. [16] presents a short evaluation of the impact of their HIDS implementation. The solution collects system calls type and arguments, and analyzes traces with a Marklov model. The authors train the model with IDEVAL dataset and some locally generated data. They admit their solution adds a noticeable time overhead, which though does not significantly impact the overall time performance of the system. [14] provides a more precise focus on timing impact of HIDS deployment on system time performance. They show a large gap of time overhead, from 3% to 77%, depending on the type of executed program. However, data are not well representative for our target system. Firstly, the study is from 2004: hardware has evolved and so time performance did. Secondly in the authors' implementation, the monitored workload is a virtual machine hosted by a type-II hypervisor. A type-II hypervisor runs inside an OS; it requires high computing power. This system architecture is not particularly adapted for embedded systems. For our mixed-criticality system instead, we use a type-I hypervisor, which directly manages hardware resources.

In our literature research, all machine-learning based HIDS implementations we found are

dedicated for deployment for generic computers running rich operating systems such as Linux or Windows. For instance, the main open-source datasets correspond to trace of execution on desktop PC: e.g. ADFA-LD [4, 11] dataset on Linux OS and KDD99 [19] on Solaris OS.

3 Methodology

We propose a methodology to evaluate the impact of intrusion detection based on system call monitoring on a mixed-criticality system with real-time constraints. The required system call monitoring causes an additional delay in the kernel at system call handling, which induces two consequences: an impact on workload performance as well as an impact on system security.

3.1 Impact on Performance

A mixed criticality system is composed of several workloads corresponding to diverse criticality levels. We distinguish two main types of workload:

- critical: software under deterministic constraints. E.g. Avionics control loops.
- non-critical: software with no deadlines. E.g. Linux OS.

3.1.1 Critical Workload

To use system call tracing in a critical workload, the additional overhead should be added to Worst Case Execution Time (WCET) of this workload. The additional delay due to system call tracing is easy to identify. We can indeed calculate the maximal overhead duration with WCET analysis; the precise tracing overhead can be estimated for each system call in the kernel. Thus, we can compute the worst case overhead, corresponding to the worst system calls combination for the analyzed software.

3.1.2 Non Critical Workload

WCET analysis does not make much sense for non-critical workloads such as Linux OS: it would either be too pessimistic or non doable. We instead aim to analyze an average time overhead. However, because of the large diversity of software, it is not possible to get a representative single average value.

3.2 Impact on System Security and Dependability

In addition to the performance impact, the system call tracing time overhead located in the kernel can potentially compromise the whole system, by breaking time isolation between workloads. A workload raising a huge amount of system calls induces indeed a non-negligible delay in the kernel.

Depending on the kernel implementation, we consider two scenarios:

1. system call handling can be preempted by context switch.
2. system call handling can not be preempted by context switch.

In the first case, even though the application induces a huge delay due to system call tracing, it is stopped by the tick interrupt, responsible for context switching; the system call is possibly aborted. Thus, time isolation is respected.

In the second case, the application can extend slightly its time window, at the expense of other workloads: time isolation is not respected anymore. An example of such issue consists in two applications running on the same single CPU core. The first is a non-critical

workload, while the second is a hard real-time workload with firm deadlines. If the non-critical application runs huge amount of system calls, there are high chances that context switch tick interrupt is raised during system call handling. The kernel waits for system call to return to the application before switching context to the next workload. Thus, we can assess the maximal time overhead per context-switch to be the length of system call handling, including system call tracing. As the system is designed for guaranteeing time isolation without tracing, the time window of the non-critical application can be extended to a maximum of $t_{overhead_{max}}$. After a certain amount of context switches, the critical application possibly fails missing its deadline. Thus, we need to define the overall time overhead induced by system call instrumentation for typical workloads, to then be able to evaluate how it could compromise the system.

3.3 Estimating the Tracing Time Overhead

We propose to compute an average time overhead at runtime for a set of representative non-critical workloads, with the following approach. The total time overhead t of an application running a sequence of N system calls corresponds to $t = \sum_{i=1}^{N} t_{syscall_i}$. The quantity t depends on the number of system calls and on their type. Firstly, we count N the amount of system call traces for every workload. Depending on the context of execution, the quantity N can vary for two different executions. Secondly for a set of well used system calls, we compute the time overhead caused by tracing. We can expect a small variation of tracing time between different system calls. The difference is indeed due to variations in the quantity of traced data caused by the number of parameters and return value. Finally, we can estimate a range of possible values for the total tracing time overhead t, so that $t_{min} \leq t \leq t_{max}$ with $t_{min} = \frac{t_{tracing_{min}}}{T} * N$ and $t_{max} = \frac{t_{tracing_{max}}}{T} * N$. T is the total time of execution, without tracing. We can not precisely measure this value: program execution varies, depending on the context of execution and regardless of system call tracing feature. Therefore for a single workload execution, system call tracing time must be associated with total execution time. Instead of T, we can measure T' so that $T' = T + t$. To finally evaluate the impact of system call tracing, we will use the ratio $r = \frac{t}{T'-t}$.

4 Experiment

4.1 System Design

4.1.1 Overview

The figure 1 details the system architecture. We run the experiment on PikeOS [8], a commercial certified mixed-criticality OS. It can be used as a real-time OS (RTOS) to run native applications. As a type-I hypervisor, it can also host more complex workloads such as complete OS. Every hosted workload is attributed levels of safety and security, where resource partitioning enforces isolated execution. A diverse range of workload types is supported (ARINC 653, Linux, POSIX, AUTOSAR, etc.). PikeOS software complies with safety standards, such as DO-178B (Avionics), EN 50128 (Railways).

The system call tracing driver stores system call traces in a per CPU core trace buffer and returns them upon request. For each system call raised, two traces are stored. They correspond to the entry and exit hooks inserted in every system call handler. Once a system call is received, the entry hook extracts the system call type and its parameters and writes the first trace into the corresponding trace buffer. Before returning from the handler, the

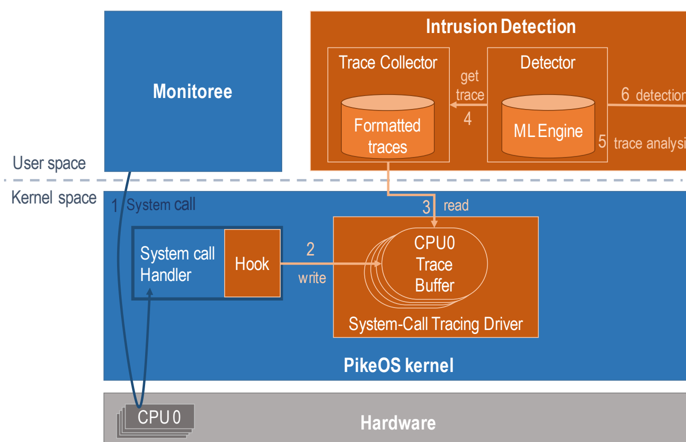

Figure 1 System architecture

exit hook gets the return value and stores the second trace into the trace buffer.

At user-level, the Monitoree, defined in 4.2.2, is the monitored workload. The IDS software is composed of two main user applications. Firstly, the detector contains the analytics model to identify a trace as footprint of an intrusion or as normal activity. This part is not yet implemented and will be developed in further work. Secondly, the trace collector collects traces by communicating with the system call tracing driver. It formats and stores the traces locally until read request from the detector.

4.1.2 System Call Tracing within PikeOS

The system call tracing process goes through the following steps (see figure 1):

1. The monitoree raises a system call.
2. The OS kernel handles the system call. From events raised by hooks in the handler, system call information are stored in the trace buffer of the initiating CPU core.
3. At some point, the trace collector requests traces to the system call tracing driver. The driver reads the trace buffer and returns traces to the trace collector. Traces are then formatted and stored waiting for request from the detector.
4. The detector requests a trace to the trace collector.
5. The trace is analyzed with the machine-learning engine.
6. In the case where an intrusion is detected by the engine, a signal is raised (e.g. an alarm message is printed on the console).

Groups of operations (1,2), (3), (4,5,6) can run in parallel. Operations inside the groups are sequential.

The tracing feature induces an additional delay for system call handling in the kernel. It corresponds to the entry and exit hooks inserted in system call handlers. The hook consists mainly in a memory copy operation into the CPU trace buffer.

For the sake of clarity, we isolate the monitoree on a single dedicated CPU core (core 0). The trace collector runs on a distinct core (core 1).

4.1.3 Limitations of the Implementation

For now, the solution does not monitor hardware virtualized guest OS. Such workload indeed directly handles system calls, without any intervention of the hypervisor. Critical instructions raise hypervisor calls and are trapped in the kernel. Our solution could be adapted to support hardware-virtualized workloads, by tracing these calls in the hypervisor kernel.

4.2 Setup

4.2.1 Test Environment

For the experiment we use the NXP board QorIQ LS1043A. Time is measured with CNTVCT_EL0 CPU core counter. It corresponds to the virtual count for execution level 0 (user mode) of the running core and is accessible from user workloads. The resolution of this counter is 40 ns. A timestamp is defined in the kernel for every trace before it is stored in the trace buffer.

In our tests, we measure the time for running a workload, by printing the counter value just before and after its execution. We get the unitary time for system call tracing in the kernel using the same method.

4.2.2 Tested Applications

We selected diverse applications from native and POSIX applications, as well as paravirtualized Linux. Native and POSIX servers are stimulated with a Linux client running `ping` process. For every test, we run measurements only once the system has booted.

Native applications are C programs running directly on PikeOS, using specific libraries. Ping server is a ICMP echo server which returns echo replies upon request. Shared memory client/server application corresponds to two applications, depending on runtime mode. Both applications share a memory buffer to exchange 32 text messages that are emitted by the server and read by the client, using synchronization mechanisms.

The POSIX application inetd is a server daemon, which relies on a widely used TCP/IP stack called lightweightIP.

The Linux workload corresponds to a paravirtualized commercial embedded Linux for PikeOS, called ElinOS [7]. Our selection gathers well used processes and common performance benchmarks. Table 1 lists command lines for all Linux applications.

Table 1 List of Linux workloads

Workloads	Command line
unixbench	`unixbench`
dd	`dd if=/dev/zero of=FILE count=CNT bs=BS`
ps	`ps -ef`
find	`find / > /dev/null 2&>1`
netserver	`netserver -4 -L ADDRESS -D -d`
netperf	`netperf -H NETSERVER`
Pacman	`pacman`
gzip	`gzip FILE`
iperf	`iperf -s`
ping	`ping ADDRESS -c 20`

5 Results

5.1 Reliability of Measurements

Every test is reproduced five times. Average values of time and amount of collected system calls traces are represented in table 2. To evaluate the variability of results, the standard deviation is calculated and compared to the average value. Variability of system call count measurements is high in two situations:

- for workloads which raise few system calls; e.g. native ping server and empty Linux OS.
- for short execution time; e.g. ps and dd Linux processes.

In this second category, time measurement is also less precise (approximately 7% of fluctuation).

5.2 System Calls in Workloads Selection

The table 2 shows that the amount of system calls varies, from 23 system calls to more than 200,000 for one second of workload execution, excluding tracing time. The results illustrate that native applications, even with networking, raise very few system calls, comparing to POSIX inetd daemon and Linux workloads.

Table 2 List of measurements for tested workloads

Workloads		time (s)	calls/s	var (%)
Native	ping server	60.45	23.2	41.16
	ping server (busy)	63.98	552.0	9.60
	sh.mem. (client)	31.47	410.0	0.00
POSIX	inetd (no requests)	30.08	10,508.0	2.75
	inetd (ping requests)	30.06	11,598.2	0.39
Linux	empty	60.34	26.7	8.80
	dd (64 MiB)	0.11	34,000.0	9.57
	dd (128 MiB)	0.22	19,398.1	7.52
	ps -ef	0.01	225,876.8	12.10
	find /	0.31	115,113.0	5.86
	gzip (128 MiB)	3.48	382.0	2.70
	ping (20 packets)	19.03	375.5	5.58
	netserver	10.01	6,120.9	0.74
	netperf	10.02	6,289.6	0.94
	iperf (server)	10.02	6,196.5	0.87
	iperf (client)	10.04	6,170.6	1.11
	pacman	20.00	4,129.5	2.95
	pacman	30.00	4,687.3	0.87
	unixbench	1,677.24	4,799.5	0.14

5.3 Unitary System Call Tracing Overhead

The granularity of the hardware counter does not allow high precision; with the 40 ns threshold, we count less than 20 ticks per unitary tracing time overhead. Nevertheless, because of the high variability in collected values (around 20%), we can still define a range of

Table 3 List of tracing overhead for a selection of system calls

System call	Min (ns)	Average (ns)	Max (ns)	var (%)
MMAP	240	391	760	20
IPC	240	399	760	19
Thread Yield	240	343	720	24
Thread Sched.	240	362	640	19
Sleep	240	382	760	23

time overhead for a set of common system calls. We measure tracing overhead for the system calls described in table 3. They indeed appear as good candidates for intrusion detection as they have a direct impact on system execution; i.e. workloads scheduling, memory access, inter-thread communication.

- Memory mapping (MMAP): maps a number of pages from one workload's address space to another.
- Inter-process communication (IPC): sends a message to a thread and receives an IPC message from another.
- Thread Yield: forces the running thread to yield the CPU.
- Thread Scheduling: exchanges thread priority and execution time window.
- Sleep: suspends the calling thread for an amount of time.

For every system call, we run 1000 measurements. For a single system call, tracing time can take a wide range of values between 240 ns and 760 ns: minimal, average, and maximal values are respectively equal to 240 ns, 376 ns, and 760 ns.

5.4 Interpretation of Results

Table 4 Time overhead for tested workloads

Workloads		Time overhead ratio		
		min	average	max
Native	ping server (busy)	1.32E-04	2.08E-04	4.20E-04
	sh.mem. (client)	9.84E-05	1.54E-04	3.12E-04
POSIX	inetd (ping requests)	2.78E-03	4.36E-03	8.81E-03
Linux	empty	6.41E-06	1.00E-05	2.03E-05
	dd (128 MiB)	4.66E-03	7.29E-03	1.47E-02
	ps -ef	5.42E-02	8.49E-02	1.72E-01
	find /	3.63E-03	5.68E-03	1.15E-02
	gzip (128 MiB)	9.17E-05	1.44E-04	2.90E-04
	ping (20 packets)	9.01E-05	1.41E-04	2.85E-04
	netperf	1.51E-03	2.36E-03	4.78E-03
	iperf (server)	1.49E-03	2.33E-03	4.71E-03
	pacman	1.12E-03	1.76E-03	3.56E-03
	unixbench	1.15E-03	1.80E-03	3.65E-03

According to the definition of security impact provided in section 3, we compute the time ratio, $r = \frac{t_{tracing}}{T_{tot} - t_{tracing}}$. $t_{tracing}$ is in the range $[t_{min}; t_{max}]$. Let $t_{tracing}$ be a variable so

that $r = f(t_{tracing})$. r is an increasing function between $\frac{t_{min}}{T_{tot}-t_{min}}$ and $\frac{t_{max}}{T_{tot}-t_{max}}$. We show minimal, average, and maximal values of r in table 4.

r values can vary significantly in function of the workload from 10^{-5} to 17%. Thus, we can not easily define a factor to predict time overhead in function of simple parameters such as time of execution and amount of static code instructions. The time overhead depends on the type of system calls and context of execution (especially for Linux virtual machines).

As seen in section 5.2, because the amount of system calls for native applications is very low, the time overhead is negligible. More complex workloads, e.g. Linux and POSIX programs, present much higher time overhead. `ps` Linux process spends particularly the longest time for tracing system calls (5% to 17% of additional time). The majority of the selected workloads correspond to a ratio r in the order of 10^{-3}, which represents a tracing time overhead of 60 ms for one minute of program execution time. The overhead affects workloads time performance, but remains reasonable in the majority of cases.

6 Discussions

6.1 Comparison of Tracing Impact with Previous Work

[14] implements a HIDS for monitoring a Linux virtual machine running on a type II hypervisor called User-Mode Linux (UML) Monitor. The authors compute total time overhead of system call monitoring for three well used Linux processes: ps, find, and ls. The overhead corresponds to the whole implementation, system call tracing and detection. Their results point already a wide diversity of time overhead, from 3% with `ps -ef` to 77% for `find / > /dev/null 2&>1`. Our implementation shows a much reduced overhead: average values are 8% in the first case and less than 1% for the second operation.

Another approach [16] analyzes the occurrences of system calls in workloads. From the high throughput of system calls processed by their solution, the authors deduce a visible but negligible impact at runtime: they argue that their HIDS can process from 12000 to 22000 system calls per seconds when common operating system's services usually run around 2000 system calls per second (six to ten times lower). This argument seems questionable for modern virtual machines, in light of our results; in our test, many of the applications raise more than 4000 system calls per second (Pacman, dd, netserver, etc.).

6.2 Security Impact

As highlighted in section 3.2, in the case where the kernel implementation can not preempt system call handling, time isolation is compromised in the system. According to our measurements, a workload can extend its time window to maximum $t_{overhead_{max}} = 760$ ns between two context switches.

6.2.1 Workloads Context-Switch Rate Influence

The risk of system call tracing impact on system security increases by reducing the context switch period. The shorter the time period for an application running system calls is, the more probable a context switch to be delayed by system call tracing is. Table 5 shows an example of worst case execution, where every context switch is delayed by a system call. The system runs two workloads, one with real-time constraints and a non-critical workload raising continuously system calls. This second software corresponds to a Linux thread repeating the command `ps -ef` in an infinite loop. In function of the context switch rate, the table shows

average amount of system calls raised by the monitored application between two context switches and the final tracing delay for 1 s of non-critical workload cumulated execution.

Table 5 Influence of the context switch period on tracing overhead

Context Switch Rate	Call/period	Overhead (1s execution)
100 μs	2.26E-04	172 ms
100 ns	2.26	8 μs

This vulnerability could be exploited by an attacker running exclusively system calls from a non-criticalprogramd to compromise availability of a critical program.

6.2.2 PikeOS Real-Time Hypervisor Use-Case

PikeOS kernel is non-preemptive. Thus, context switch can not directly preempt system call handling in the kernel. Nevertheless, to control time spent in the kernel, it includes preemption points, notably for long operations and before returning to user-space. Some long system call handlers, such as creating or destroying a task (above 10 μs in average), can include preemption points; though most of them do not, because of their speed (generally less than 1 μs). Hence, tracing impact described in section 6.2.1 also applies to PikeOS real-time hypervisor.

6.3 Considerations for the Detection Engine Implementation

Intrusions in mixed-criticality systems most likely target non-critical workloads: unlike closed critical workloads, they usually contain vulnerabilities. The attacker aims either to steal data from the system or to disturb its execution. In both cases, she eventually has to interact with the system (OS kernel, other workloads) from the non-critical software. Thus, our strategy consists in monitoring all workloads using a single detector. The detector would analyze system call generic information (parameters, type, etc.), correlated to the thread 's criticality level and its execution context: e.g. the running CPU core, and interactions with other workloads. As soon as a thread is identified as a threat, it should be isolated from other system entities. We also consider rollback mechanisms, especially for affected critical workloads to restore them in a safe state.

6.4 Other Promising Features for ML-based HIDS

To detect intrusions in a mixed-criticality system with Machine-Learning, we also consider further interesting features.

6.4.1 Performance Counters

Performance counters tracing seems a promising approach to detect intrusions with machine-learning, as highlighted by [1] and [6]. Tracing performance counters's values would have no direct impact on system real-time constraints as they can be read from user mode, without kernel intervention. Tracing can then be done in a separate monitor workload, running on same CPU core as the monitored workload, subject to the compliance of the monitor with system real-time constraints. This must be ensured, as well as for every workload, by the OS kernel.

6.4.2 Branch Tracing

Branch instructions provide relevant information regarding security, e.g. to guarantee program integrity with CFI (see section 2). Although we could not find any publication on this topic, we consider branch instructions as interesting features to investigate for machine-learning based HIDS. Tracing branch transfers can be done either in OS kernel software or in the hardware: e.g. ARM CoreSight [3] debugging hardware provides this tracing feature. Hardware implementation is preferable for performance speed-up and time isolation of workloads. With appropriate access rights, the monitoring application can indeed collect traces directly from user mode, without systematic interactions with the kernel.

6.4.3 Network Monitoring

Research of the last two decades has well investigated network intrusion detection systems (NIDS), that are even proposed in main IT security solutions on the market. Network monitoring traces network packets transiting from and to the monitored application. It relies on same principle of system call tracing, updating the tracing location from system call handler to the network driver. As well as with system call tracing, the overall timing impact should be non negligible.

7 Conclusion

We evaluated the impact of system call instrumentation for deploying intrusion detection in embedded mixed-criticality systems, through a concrete implementation based on PikeOS hypervisor. The implementation shows a reasonable time overhead for the majority of workloads. However, because of new delays induced for tracing in the OS kernel, time isolation between workloads is not guaranteed anymore. In further work, we plan to develop a system call based HIDS to monitor a mixed-criticality system, which enforces time isolation. We will research adapted HIDS architectures for our system context. We will investigate how system call flows from different workloads can be combined for a reliable detection.

References

1 Muhamed Fauzi Bin Abbas, Sai Praveen Kadiyala, Alok Prakash, Thambipillai Srikanthan, and Yan Lin Aung. Hardware performance counters based runtime anomaly detection using SVM. In *TRON Symposium (TRONSHOW)*, 2017.

2 Jim Alves-Foss, W. Scott Harrison, Paul Oman, and Carol Taylor. The MILS architecture for high-assurance embedded systems. In *International Journal of Embedded Systems*, 2005.

3 ARM. *CoreSight Technical Introduction*, 2013. White Paper: ARM-EPM-039795.

4 Gideon Creech and Jiankun Hu. Generation of a new IDS test dataset: Time to retire the KDD collection. In *IEEE Wireless Communications and Networking Conference*, 2013.

5 M. T. Elgraini, N. Assem, and T. Rachidi. Host intrusion detection for long stealthy system call sequences. In *Colloquium in Information Science and Technology*, 2012.

6 David Fiser and William Gamazo Sanchez. Detecting attacks that exploit meltdown and spectre with performance counters. `https://blog.trendmicro.com/trendlabs-security-intelligence/detecting-attacks-that-exploit-meltdown-and-spectre-with-performance-counters/`, 2018. [Jun. 05, 2019].

7 SYSGO GmbH. Elinos embedded linux webpage. `https://www.sysgo.com/products/elinos-embedded-linux/`. [Jun. 05, 2019].

8 SYSGO GmbH. PikeOS hypervisor webpage. `https://www.sysgo.com/products/pikeos-hypervisor/`. [Jun. 05, 2019].

9 W. Haider, J. Hu, , and M. Xie. Towards reliable data feature retrieval and decision engine in host-based anomaly detection systems. In *IEEE 10th Conference on Industrial Electronics and Applications (ICIEA)*, 2015.

10 Mohamed Hassan. Heterogeneous MPSoCs for mixed criticalitysystems: Challenges and opportunities. In *IEEE Design and Test Magazine*, 2017.

11 Jiankun Hu. AFDA-LD dataset webpage. `https://www.unsw.adfa.edu.au/unsw-canberra-cyber/cybersecurity/ADFA-IDS-Datasets/`, 2013. [Jun. 05, 2019].

12 Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

13 Koucham, T. Rachidi, and N. Assem. Host intrusion detection using system call argument-based clustering combined with bayesian classification. In *SAI Intelligent Systems Conference (IntelliSys)*, 2015.

14 M. Laureano, C. Maziero, and E. Jamhour. Intrusion detection in virtual machine environments. In *30th Euromicro Conference*, 2004.

15 Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, A. Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

16 F. Maggi, M. Matteucci, and S. Zanero. Detecting intrusions through system call sequence and argument analysis. In *IEEE Transactions on Dependable and Secure Computing*, 2010.

17 Lee Pike, Pat Hickey, Trevor Elliott, Eric Mertens, and Aaron Tomb. Trackos: A security-aware real-time operating system. In *International Conference on Runtime Verification*, 2017.

18 The LLVM Foundation. The LLVM compiler infrastructure. `llvm.org`. [Jun. 05, 2019].

19 University of California. KDD Cup 1999 data webpage. `https://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`, 1999. [Jun. 05, 2019].

20 C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: alternative data models. In *IEEE Symposium on Security and Privacy*, 1999.