Timing Architecture Model for Embedded Systems Anomaly Detection

Peter Heller, Jürgen Mottok

Timing Architecture Model for Embedded Systems Anomaly Detection

Peter Heller and Jürgen Mottok
Laboratory for Safe and Secure Systems (LaS³)
Technical University of Applied Sciences Regensburg
93053 Regensburg, Germany
{peter2.heller, juergen.mottok}@oth-regensburg.de

***Abstract*—By using execution timing behaviour to discover anomalies, embedded systems can be monitored at various architectural layers. Different methods for deducing sane system execution behaviour based on available event or timing data are proposed in the current literature about security-related anomaly detection of embedded systems. With our work, we evaluate several strategies and discuss problems with accessible metrics and architectural components used for feature development. An embedded system's architecture layers serve as the basis for a common classification scheme that makes it possible to combine timing- and event-based metrics into a single timing architecture layer model. Then, using metrics and architecture components, our suggested model is applied to several anomaly detection techniques and utilized to compare existing methods. Our mapping leads us to the conclusion that most detection models are restricted to single system layers (i. e., communication or application code) and use a small number of accessible architecture levels. Our existing model allows us to combine various time and event metrics, but we also want to develop new features for embedded anomaly detection that can be used across all system layers (code, scheduling and communication).**

***Keywords*—anomaly detection, time series data, embedded systems software, architecture layer model**

I. Introduction

Due to ongoing regulatory developments, preventive measures against malware will be an integral part of the product lifecycle within the European Union in the foreseeable future. With the two directives 2023/1230 [1] and NIS 2 [2] issued by the European Parliament, manufacturers and suppliers will be obliged to implement appropriate security measures for their products. One objective of these regulations is to counteract tampering and to prevent or at least mitigate malicious intrusion of safety-relevant control systems. Real-time capable systems exhibit measurable side effects on the remaining system when subjected to changes regarding tasks or other critical code sections (see [3]). Analysing execution timing offers a way to implement anomaly detection by evaluating changes to known system behaviour based on a selection of indicators at different architecture levels, and thus to identify suspicious activity.

Available research on anomaly detection provides us with various approaches for detecting changes in system behaviour. The problem with current solutions, however, is that it is difficult to compare the different methods with one another. Some approaches use event-based metrics from the operating system or a communication stack, while other models use timing measurements at the instruction level to detect deviations from the specified behaviour. Because no common model exists to date, one objective of this paper is to derive a suitable classification to map available metrics from the literature into a single scheme. Scheduling theory for embedded real-time systems uses models and architecture frameworks to characterize the execution behaviour. By combining an architectural model and suitable notations for different workloads from scheduling theory, we introduce a model that can combine time- and event-based metrics from different architecture levels in a common representation. We want to use this representation to merge existing features for anomaly detection in a hybrid approach that leverages data based on a collection of metrics captured from different architecture levels.

In summary, the focus of this work is to address the following research question:

Q1. Can available timing and event metrics for embedded systems anomaly detection be mapped into a common classification scheme based on their system architecture?

The following Section II starts with a general introduction of the literature on embedded systems anomaly detection with security related context, where we focus on RTOS based systems. We discuss the general problem of using time-series data and give a brief overview of the current state of the art. Section III then presents the timing architecture layer model for developing a hybrid approach on multiple architecture levels, and categorizes existing literature from Section II based on the aforementioned model. The final Section IV briefly summarizes the key findings and the next steps in our research process.

II. Related Work: Anomaly Detection using System Timing and Event Metrics

The current literature on embedded systems presents a challenge due to the variety of methods employed, architectural levels involved, and underlying metrics applied in anomaly detection for time-series data. To improve our understanding, we want to focus on two important aspects based on anomaly detection literature: The applied metrics (runtime-data and events) and architectural components of currently available approaches for embedded systems. First, how available metrics are used and what type of data those metrics are based on. Secondly, which layers of the system architecture were used and provided to be useful for the development of new anomaly detection features. To narrow the available literature on anomaly detection using time-series data, two selection requirements were used: (*i*) The Data source for anomaly detection is based on embedded system traces. (*ii*) Selected publications should have a security context. Context in this regard implies that the objective for anomaly detection is security-relevant for the system, or that the verification process of the created anomaly detection mechanism is targeting tampering of the target system. The purpose of this

selection is to ensure that the range of relevant metrics for the subsequent comparison have common ground.

In general, the available literature can be split into two different categories based on the applied metrics: *event-based* and *time-based* approaches. Events are observable changes in system state and are generated during system execution. The goal of recording events is to determine if a sequence of events, the frequency of events or temporal dependencies show measurable deviations from normal behavior. Depending on the architectural origin of an event (i. e application- or instruction-level), it can be observed either from software or hardware. Events can be described as a tuple of system defined values (see [4]) denoted as $e = \langle v_1, v_2..v_x \rangle$ holding event-specific information, which in turn is highly dependent upon where the specific event is generated (network-stack, scheduler, application code, instruction-level). Optionally, a timestamp can be incorporated to denote chronological dependencies between single events for time-based approaches. In this case, the timing information needs to be obtained and processed either on the target system or measured by using an external time base. Depending on system complexity and hardware architecture, we can observe and record different event streams at varying levels of granularity regarding overhead and intrusiveness. Therefore, we can not always observe a single event, but a subset of available data.

Embedded real-time systems implement deterministic scheduling models to design and verify their intended functionality, and thus exhibit recurrent behaviour and distinct execution patterns. Three identified approaches [4]–[6] use Inter Arrival Curves (IACs) or modified modelling techniques with similar properties to infer system state based on a selection of events. IACs characterize the activation pattern of individual event streams, by limiting upper and lower bounds of event occurrences for a given time window. A combination of different activation functions based on arrival curves allows modelling the dynamic behaviour of different event-based systems. Ezeme et al. [5] and Torres [6] have shown how recurrent timing and pattern detection can be implemented for (online) system monitoring utilizing the aforementioned event-based metrics. Salem et al. [4] use a different approach by aggregating multiple event streams into sequence-based arrival curves (IACs) which are then used to derive suitable features for anomaly detection. Hoffmann et al. [7] utilize a similar approach with multiple event streams, but on a different architecture level. Their work focuses on performance metrics that are captured at CPU execution level using system performance counters. Lu and Lysecky [3] implement different models (range-based, distance-based and SVMs) utilizing only timing parameters at the lowest architectural levels to detect changes in software behaviour when subjected to different types of malware. In this case, the hardware trace port is used to extract core registers and measure timing with an external time base. By exposing several signals from the hardware trace port, cache, and pipeline effects can be observed and fed into the detection algorithm. With their hardware-based approach, they can measure execution time down to the instruction level and capture specific timing parameters they are interested in.

Given our available literature, we can apply a preliminary classification of the individual parameters. First, high-level software events, which can be captured from the operating system, a scheduler, or the communication stack of a target. Due to the nature of these events, the streams, and traces contain a high level of noise from other event sources. Second, low-level events, which are based on performance counters or hardware peripherals and allow detection of anomalies based on core execution behaviour at the deeper system levels. Third, execution time, which can be determined at different architecture levels and with different methods, but usually requires measurement. All of those metrics use different data types and originate from different system architecture levels. What is interesting to us is a) the variety of available metrics and b) the limited use of the available system architecture components, which we want to address in the following Section III. For this purpose, we want to introduce our timing architecture model to map parameters and architecture components into a common classification scheme. We want to apply our model to develop new features for anomaly detection, allowing us to extend detection models to all available system architecture layers. To the best of our knowledge, there is no known study or paper that maps available time- or event-based metrics into a common model or scheme targeting embedded systems at the time of writing.

III. Mapping Time and Event Metrics based on Architecture Levels

System architecture layers can be used as a descriptive tool to decompose and analyse metrics within a given system, which is why we choose this approach as a tool to compare available literature. We are interested in understanding how different methodologies for embedded devices use the existing system architecture to implement effective anomaly detection with event streams and time series data. For this purpose, we based our model on an existing layering model for timing analysis of embedded real-time systems [8] and extended their work by applying a workload-based decomposition from the automotive domain [9]. This extension of the model allows us to further differentiate existing approaches below the application level, so we can map available anomaly detection methods based on architecture and metric usage.

Based on our preliminary evaluation of available literature, we found that event-based metrics are used more frequently instead of runtime measurements. Therefore, we need to consider both timing and event-based metrics during our mapping procedure. For this purpose, our current model provides two degrees of abstraction, called *layers* and *levels*, to represent architecture and timing/event properties we are interested in. The available model provides three high-level architecture layers: a *Communication Layer*, a *Scheduling Layer* and an *Application Code Layer* to represent the generic structure of an embedded system. Layers are used to represent specific key timing properties of an embedded system, such as *Core Execution Time (CET)*, *Response Time (RT)*, and *Round Trip Time (RTT)*. Levels are used to assign a source of events or timing parameters to a system component. They allow further differentiation of where different timing parameters, events, and other interesting metrics originate and where their area of effect is located inside the architecture. For a single layer, subcomponent timing is used to calculate each key parameter based on the assigned levels. For further reference, Table I shows an overview of all nine architecture levels, abstraction layers and descriptions for relevant timing and event metrics as well as the key parameter for each Layer.

TABLE I
SYSTEM ARCHITECTURE LAYER MODEL FOR DECOMPOSING TIMING AND EVENT METRICS BASED ON ARCHITECTURE ORIGIN

Name	Level	Layer	Key Parameter	Description
Instruction	1	Code	CET	Smallest measurable quantity for tracing (single instruction, pipeline events, caching, fetch)
Basic Block	2			Continuous sequence of code with a single entry and exit point (branch — jump)
(Sub) Function	3			Decomposed task job sequences (notification, locks, signals, resource access)
Top-Level Function	4			Runnables and task-jobs, workload management (job start, job stop, job dispatch)
Task, ISR	5			Task start/stop, task preemption, application specific ISR, background task jobs
CPU-Core	6	Scheduler	RT	Singlecore scheduling effects: synchronization, spinlocks, memory-management, scheduling ISRs
Processor	7			Multicore scheduling effects: synchronization, spinlocks, memory-management, scheduling ISRs
ECU	8	Comm.	RTT	Interprocessor level: bus interfaces between system internal devices (SPI, I2C ...)
Network	9			External level: Networked Signals/Events, Network-Latency, Roundtrip Time

A. Timing Architecture Layer Model

While the original layering model for timing analysis does not provide a formal notation, other works [9], [10] can be used to enhance the mapping of available events and timing parameters to the architecture layers. For our use case, we remove the lowest available level (OP-Code/Micro-instruction, L0) from the original model [8], since single machine instructions are the smallest measurable quantity we can effectively capture and process using available tracing methods.

Considering the existing model, the key parameter to the code layer is the Core Execution Time (CET) of the associated task(s) or application. The CET reflects the actual time a specific task, a function, or a sequence of instructions executes without overhead through preemption or scheduling. Any timing parameters and associated events necessary to compute the raw execution time of application code are accounted for within the first five layers of the architecture model. For this purpose, we introduce a workload-based decomposition that can be employed to partition application code of an embedded system into three different levels of granularity. Levels 3 to 5 can be illustrated as a set of tasks implementing an event queue executing requested workloads, as shown in Figure 1. Depending on the event queue, the active application task handles specific system events, that are scheduled to execute certain processing workloads (jobs). At each event execution cycle, queue entries are released and used to schedule a number of jobs for the active task. Events and timing parameters related to level 5 of our architectural model are captured in the scheduler and task loop of the active task. This level is used to observe timing and behaviour at task level granularity, i. e. when a single task is scheduled or pre-empted. Overhead through execution time for maintenance, monitoring and tracing functions, application specific ISRs and background jobs executing in the idle task are included at this level. Similarly, events (i. e. logs or syscalls) generated through monitoring, profiling and tracing are associated to this level. Level 4 consists of timing and events generated through executing scheduled workloads for a single task. This level is used to trace how workloads consisting of a single task job or a continuous sequence of individual jobs behave.

Different authors define a *job* or a *runnable* as a collection of code that performs higher level functionality (see [9], [10] and [11]). Since runnables or task jobs implement higher-order functions, we needed another level of abstraction to break down application code below this level. This would allow us to characterize functions, events, and portions of code needed to perform complex operations. For this reason,

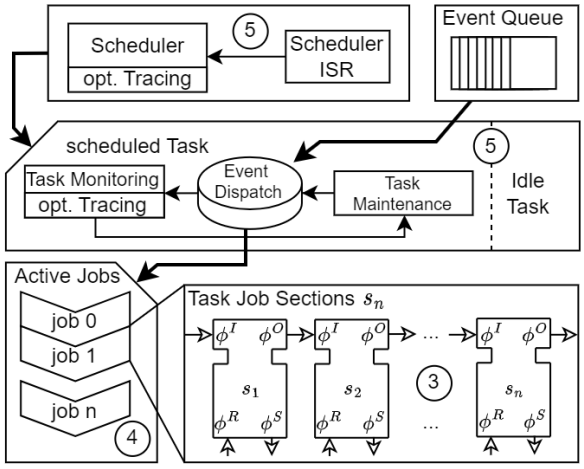

Fig. 1. Overview of a generic embedded application with architecture and event/timing components: scheduling and active task correspond to level 5, task jobs to level 4 and decomposed sections are assigned to level 3.

we introduce the definition of a *task job section*, which can be understood as a section of code that contains a sequence of instructions defined with an execution time model and signal vectors (ϕ^I , ϕ^O, ϕ^R , ϕ^S), per definition of [9]. For our use case, we want to extend this definition to be applied at the granularity of single jobs instead of the entire task, hence the naming task job section. Signals are input or output events for any given task, including the executing task, which are used to control or synchronize subsections of jobs and other runnables (tasks). Signals can generally be distinguished into two different categories: signals required for computation and signals required for synchronization. First, the input and output signal vectors (ϕ^I, ϕ^O) are computational signals because they provide the data being used and transformed during processing within a single task job activation. These signals define the interface of a given task job section, since they enforce the required input and output parameters needed for a specific section of code. The second category can be described as event signals, which are denoted as requested signals (ϕ^R) and supported signals (ϕ^S). Requested signals (ϕ^R) are required input signal vectors, that are consumed at the start of each section. This type of signal is used to provide events (i. e. notifications or data from other tasks) to a sequence of code and can be used for synchronization purposes. Supported signals (ϕ^S) are consumable events or notifications generated by the executed section of code, which are required by other tasks or subsequent task activations.

Timing based on task job sections at the granularity of available signals makes up level 3 of our architecture model.

This level allows analysing execution runtime by tracing single signals passed between tasks and the scheduler/operating system, as well as timing based on the execution model of each task job section. Generally, these signals are used to exchange data between tasks or for resource management implemented by the scheduler.

The second level (2) is used to assign metrics that can be captured at basic block granularity. In this case, any sequence of code between branch/jump instructions can be instrumented or measured. On the first level (1), the timing or activation of a single instruction is the primary feature. This type of measurement allows for single cycle granularity when capturing available metrics, either by runtime measurement of a single instruction or by observing instructions through a trace-port as an event stream. Since our mapping needs to cover timing, event and counter-based metrics, that can be observed through specialised hardware, we need to include caches, retired instructions, memory access and pipeline effects at this level.

The key parameter, within the scheduling layer, is the Response Time (RT) for the scheduled task set. This layer introduces execution time and latency from the scheduler, to account for timing variations outside of application code. Level 6 represents timing and events that are present at single-core task scheduling level. This level is used to capture timing effects introduced by scheduling and resource management handled centrally throughout the system. Runtime caused by locking mechanisms, timing overhead introduced by the scheduler, or scheduling ISR are accounted to this level. Level 7 extends the definitions of level 6 to multicore scheduling and overhead that is introduced by sharing cores, resources, and workloads. The distinction between the two levels is used to separate the impact caused by multicore processors, instead of mixing metrics together.

The final communication layer is used to introduce timing and events related to external communication and focuses on the Round Trip Times (RTT) for available interfaces. For illustrative purposes, we further differentiate between interprocessor (level 8) and external communication (level 9). At the interprocessor level, system-bus timing and packet handling events within system boundaries are handled. The external level handles events and signals leaving the system boundaries, e. g. network connections to other systems.

B. Application of the Timing Architecture Model

We introduced our timing architecture layer model to map available metrics provided by related work. Our goal was to create a common representation to compare several embedded anomaly detection techniques, based on metrics and architectural components. We saw that two publications [3], [7] implement detection models utilizing timing and events at levels 1 to 3. In addition to the lower level timing aspects, timing variation and events due to cache and pipeline behaviour are also considered in both approaches. All other works [4]–[6] are focused on high-level metrics (5,8,9) and implement their approaches without any additional input from the underlying architectural layers. Table II shows a quick outline of the different parameters used by current approaches, as well as our applied mapping.

From the existing publications, we can infer that communication and code layer parameters are being preferred as a source for different anomaly detection methods. Specific timing parameters from timing theory (CET, RT) are found in two of the available publications as part of input vectors for heuristic models or machine learning-based detectors. Event-based systems tend to use the sequence information of observed event streams at the code and communication layers to infer the state of the underlying system. Based on the architectural model, we can determine that the scheduling layer is not actively used in any available publication. While tasks are monitored, instrumented and traced at the code level, resource management, workload sharing and locking features are underutilized. For communication-based systems, the application should require heavy use of resource management and locking features, making them good candidates as features for anomaly detection in this context.

TABLE II
APPLIED SYSTEM ARCHITECTURE LAYER MODEL

Source	Used Parameter	Architecture Level	Architecture Layer
[3]	Execution Time, Cache	1,2,3	Code
[5]	Inter Arrival Curves	5,8,9	Code, Comm.
[4]	Inter Arrival Curves	5,8,9	Code, Comm.
[6]	Inter Arrival Curves	5,8,9	Code, Comm.
[7]	Slack, Interrupts and Event-Counters	1	Code

IV. CONCLUSION AND OUTLOOK

Our goal was to evaluate how anomaly detection for embedded systems is performed and what architecture components are used to implement novel detection algorithms. We introduced a timing architecture layer model for mapping publicly accessible timing and event metrics based on architecture components into a common classification scheme. The proposed model was introduced as a comparison tool, since current research uses various metrics and architecture layers to implement anomaly detection methods. We applied our model to the available anomaly detection literature, which was selected based on embedded security context, and show how different timing and event related metrics are utilised. We found that there is an under-utilization of parameters present in the scheduling layer of our architecture model. We also believe that a more in-depth comparison of time and event metrics could be a valuable contribution.

Our work targets embedded devices with strict timing requirements, like soft and hard real-time embedded systems or RTOS based systems. Since their system operation has specific requirements on execution timing and response, even slight changes can have measurable impact on the timing characteristics. For our future work, we want to evaluate whether anomaly detection on all available system layers (code, scheduler, and communication) is a feasible approach based on data available from embedded systems or whether single level approaches are sufficient. In terms of application within the security domain, another area of interest is remote attestation protocols. For our future work, we would like to determine, if time-based anomaly detection could be extended to all available system layers for attestation protocols. With our current work, we want to evaluate novel features, so we can determine which architecture component provides the best use for effective anomaly detection and parameter selection.

ACKNOWLEDGMENT

The presented work is part of the research project KRITIS Scalable Safe and Secure Modules (KRITIS³M), which is funded by the Project Management Jülich (PtJ) and the German Federal Ministry for Economic Affairs and Climate Action (BMWK) under funding code 03EI6089A.

REFERENCES

[1] Council of European Union, "Regulation (EU) no 2023/1230 on machinery and repealing Directive 2006/42/EC of the European Parliament and of the Council and Council Directive 73/361/EEC," Official Journal, 6 2023, L 165, p. 1.

[2] ——, "Directive (EU) 2022/2555 of the European Parliament and of the Council of 14 December 2022 on measures for a high common level of cybersecurity across the Union, amending Regulation (EU) no 910/2014 and Directive (EU) 2018/1972, and repealing Directive (EU) 2016/1148 (NIS 2 Directive)," 12 2022, L 333, p. 80.

[3] S. Lu and R. Lysecky, "Data-driven anomaly detection with timing features for embedded systems," ACM Trans. Des. Autom. Electron. Syst., vol. 24, no. 3, apr 2019. [Online]. Available: https://doi.org/10.1145/3279949

[4] M. Salem, M. Crowley, and S. Fischmeister, "Anomaly detection using inter-arrival curves for real-time systems," in 2016 28th Euromicro Conference on Real-Time Systems (ECRTS), 2016, pp. 97–106.

[5] M. O. Ezeme, Q. H. Mahmoud, and A. Azim, "Hierarchical attention-based anomaly detection model for embedded operating systems," in 2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2018, pp. 225–231.

[6] R. T. Labrada, "Multi-signal anomaly detection for real-time embedded systems," 2020, master thesis, University of Waterloo.

[7] J. L. C. Hoffmann, L. P. Horstmann, and A. A. Frohlich, "Anomaly detection in multicore embedded systems," 2019 IX Brazilian Symposium on Computing Systems Engineering (SBESC), pp. 1–8, 11 2019.

[8] P. Gliwa, Embedded Software Timing: Methodik, Analyse und Praxistipps am Beispiel Automotive. Springer Fachmedien Wiesbaden, 2021. [Online]. Available: https://doi.org/10.1007/978-3-658-26480-2

[9] M. Deubzer, "Robust scheduling of real-time applications on efficient embedded multicore systems," Ph.D. dissertation, Technische Universität München, 2011.

[10] M. Alfranseder, "Efficient and robust dynamic scheduling and synchronization in practical embedded real-time multiprocessor systems," Ph.D. dissertation, Technische Universität Clausthal, Clausthal, Dec 2016. [Online]. Available: https://doi.org/10.21268/20161207-083032

[11] S. K. Baruah, M. Bertogna, and G. C. Buttazzo, Multiprocessor Scheduling for Real-Time Systems, 2015th ed., ser. Embedded Systems. Springer International Publishing, Apr. 2015.