AutoSec: Multidimensional Timing-Based Anomaly Detection for Automotive Cybersecurity

Milan Tepić*, Mohamed Abdelaal*, Marc Weber†, Kurt Rothermel*
*Institute of Parallel and Distributed Systems, University of Stuttgart, † Vector Informatik GmbH
Email: *first.last@ipvs.uni-stuttgart.de, †first.last@vector.com

***Abstract*—Nowadays, autonomous driving and driver assistance applications are being developed at an accelerated pace. This rapid growth is primarily driven by the potential of such smart applications to significantly improve safety on public roads and offer new possibilities for modern transportation concepts. Such indispensable applications typically require wireless connectivity between the vehicles and their surroundings, i.e. roadside infrastructure and cloud services. Nevertheless, such connectivity to external networks exposes the internal systems of individual vehicles to threats from remotely-launched attacks. In this realm, it is highly crucial to identify any misbehavior of the software components which might occur owing to either these threats or even software/hardware malfunctioning.**

In this paper, we introduce AutoSec, a host-based anomaly detection algorithm which relies on observing four timing parameters of the executed software components to accurately detect malicious behavior on the operating system level. To this end, AutoSec formulates the task of detecting anomalistic executions as a clustering problem. Specifically, AutoSec devises a hybrid clustering algorithm for grouping a set of collected timing traces resulted from executing the legitimate code. During the runtime, AutoSec simply classifies a certain execution as an anomaly, if its timing parameters are distant enough from the boundaries of the predefined clusters. To show the effectiveness of AutoSec, we collected timing traces from a testbed composed of a set of real and virtual control units communicating over a CAN bus. We show that using our proposed AutoSec, compared to baseline methods, we can identify up to 21% less false positives and 18% less false negatives.

***Index Terms*—Anomaly Detection, Real-time Operating Systems, Timing Constraints, DBSCAN Classifiers**

I. Introduction

The automotive industry is a highly demanding industry with high level of standards to ensure the safety and security of the passengers followed by their comfort and entertainment. In general, digitization within modern vehicles is broadly increasing at the same fast pace as digital communication with the environment [1]. Accordingly, several car-to-many (Car2X) communication protocols have broadly been employed to enhance the safety of passengers while enriching their drive comfort. Such protocols harness the wireless medium for sharing valuable information about the surroundings, road cases, safety warnings, etc. For instance, the reaction time of a human driver is ranged between 0.75s to 1.5s [2]. If individual vehicles are able to communicate with each another, such reaction times can be highly reduced, thus avoiding road traffic accidents. In this context, Car2X protocols can assist drivers through offering warnings about harsh braking situations, stationary roadside breakdowns, and emergency vehicles. Hence, there exists an immense need for sharing knowledge between the neighboring vehicles and the roadside infrastructure to improve the safety systems through making optimal decisions in a timely manner.

This work is partially supported by the German Federal Ministry of Education and Research (BMBF) grant 01DH17059.

In fact, the main side effect of the interconnectivity between modern vehicles and their surroundings is exposing the internal systems, including safety-critical sub-systems, of such vehicles to external threats from malicious cyberattacks. For instance, Miller and Valasek [3] carried out an experiment to remotely attack a modern vehicle. Specifically, they were able to completely overtake the control of the vehicle through breaching into the vehicle's network together with interfering the controller area network (CAN) messages. The primary security concern of this experiment is their ability to remotely override the driver's action via the wireless network despite being many kilometers away from the vehicle. Indeed, this experiment opened the door for exerting more efforts in the realm of automotive cybersecurity. Besides, short range wireless interfaces, e.g. Bluetooth, and wired ones still offer diverse attack surfaces. Therefore, much attention has been given to the development of precise network-based anomaly detection algorithms for the sake of discovering any malicious behavior of the intra-vehicle communication networks.

Despite achieving promising results in terms of the detection precision and false positives, advanced attacks may penetrate the network to deliberately inject malicious code or even to disrupt the normal execution [4]. In addition, the legitimate software components—defined by the car manufactures during the design time—may exhibit abnormal behavior due to software/hardware malfunctioning. Accordingly, the concept of solely observing the communication network is typically not sufficient for ensuring high-level of safety and security. Such a conclusion opened the door for developing several host-based anomaly detection algorithms on the operating system level [5], [4], [6]. In this context, observing the computation time of the software components has been proposed to detect unexpected or suspicious behavior of such components [6]. In Section II, we introduce an experiment which shows that monitoring only the computation time and ignoring other important timing parameters typically deteriorates the detection accuracy, i.e. large number of false positives and/or false negatives. Therefore, a challenge of detecting possible breaches in the various software components of the vehicles emerges.

To tackle this challenge, we introduce AutoSec, a multidimensional anomaly detection algorithm which relies on observing the timing parameters of the real-time software components running on various electronic control units (ECUs). To check for malicious behavior of a certain component, AutoSec considers not only the computation time of such a component but also the start-to-start (S2S) time between subsequent instances of execution, the number of preemptions, and the preemption time. In this manner, AutoSec draws a complete picture about the circumstances of each execution, thus making optimal decisions about the abnormal executions. To this end, AutoSec exploits clustering algorithms for defining—at the design time—a timing model which is then used during the runtime for making decisions about each execution. AutoSec can be used in combination with an AUTOSAR [7] classic-based ECU software where it can observe the timing parameters on two different granularity levels, including the coarse-grained task level and fine-grained runnable level (cf. Section III). It is worth mentioning that the main scope of AutoSec is to evaluate our proposed feature selection method for automotive industry standards as tracking only the computation time does not achieve the required accuracy level. To the best of our knowledge, AutoSec is the first host-based anomaly detection algorithm leveraging four timing parameters to precisely discover malicious behavior of the ECU software on two granularity levels.

In detail, the paper provides the following contributions: (1) We define an architectural framework and multi-step process for the detection of any malicious behavior of the various software components. (2) We present an algorithm that, as a first step, defines a timing model of legitimate executions performed during the design time. (3) We perform a comparative study to select a clustering algorithm which best fits with our defined requirements. (4) We devise a novel hybrid clustering algorithm in which DBSCAN [8] is initially employed as a preprocessing step for removing outliers and generating a number of clusters. Subsequently, the K-Means method [9] together with the Silhouette method [10] are exploited for discovering possible sub-clusters within each cluster. (5) We define an algorithm that, in a second step, leverages the principal component analysis for reducing the dimensions, before making decision during the runtime about the various executions. (6) We present a proof-of-concept implementation and evaluation in a pseudo real-world scenario. To this end, we created a testbed of a real ECU, connected to two virtual ECUs through a CAN bus. In these experiments, we collected over 1.5 GBytes of timing traces on both the task level and the runnable level. We show that our proposed AutoSec algorithm, compared to baseline methods, achieves higher level of detection accuracy measured in terms of the precision, recall, and false positive rates.

II. Related Work

In this section, we review the most salient related work in the realm of anomaly detection in real-time systems with highlighting the novelty of our proposed solution. In fact, several research activities which leverage different types of observables, including the power profile, the system calls, and the timing parameters, have been performed to identify the malicious behavior of the software components [5], [6], [4]. For instance, Abbas et al. [5] propose a hardware-based anomaly detection method which relies on observing the power profile of the executed software components. To this end, they utilized an ECU with a built-in feature of precisely measuring the power consumption. The anomalistic behavior in the system is detected if the power consumed during the runtime deviates from the profile recorded during the design time. However, this approach suffers from the lack of hardware support were not all ECUs posses highly-precise and fast enough energy consumption meters required to minimize the number of false alarms. Along a different line, Tong et al. [4] introduce LogSed, an anomaly detection method which monitors the system calls, i.e. the order of tasks execution. To do so, LogSed generates a time-weighted control flow graph that captures healthy execution flows of each software component. In this case, an anomaly is detected if the execution order differs from the generated graph. Nevertheless, relying on the system order is extremely risky since the executions of the software components may diverge at a certain point, thus leading to distinct execution paths.

Aside from the power profile and system calls, Yoon et al. [6] propose SecureCore, an anomaly detection algorithm which relies on monitoring the execution time of the various software components. The core idea behind SecureCore is to estimate the probability distribution of a set of execution times collected during the design time. During the runtime, an anomaly can be detected whenever the execution time of the currently-running task does not fall into the Gaussian distribution determined in the design time. Nevertheless, relying solely on the execution time of the software components does not reveal enough information about the circumstances of each execution, thus negatively affecting the detection accuracy. To prove this claim, we carried out an experiment of repeatedly executing a certain task. During this experiment, we adopted the same rules defined by SecureCore together with observing the duration between subsequent executions, i.e. the S2S time.

Figure 1a demonstrates the probability density function of the execution times collected while running the examined task. Figure 1b depicts a two-dimensional scatter plot for the execution time and the S2S time. As depicted in figure 1a, the execution time around 212 μs falls between two Gaussian distributions, thus it is considered as an anomaly according to SecureCore. However, through taking a closer look into Figure 1b, we found that the data points representing this long computation time have a normal S2S time of 10.05ms and are not falling apart from other data points in the 2D space. Conversely, there exists a single point—located at the bottom left of the 2D space—which has to be classified as an outlier. However, its execution time (roughly equal to 207 μs) lies on one of the Gaussian distributions defined by SecureCore. This implies that SecureCore may mistakenly classify such executions due to considering only the execution time. To overcome this problem, we propose AutoSec, a timing-based anomaly detection algorithm which mainly relies on observing four timing parameters, including the execution time, the S2S time, the number of preemptions, and the time of preemption. Considering these parameters enables the detection algorithm to deeply understand the context of each execution, thus minimizing the false alarms forwarded to the system.

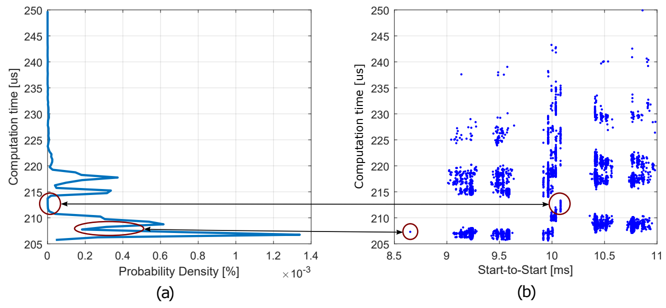


Fig. 1: Example showing that computation time is not sufficient for detecting anomalistic behaviors

III. System Overview & Preliminaries

A. System Architecture

In this section, we explain the system model together with our assumptions. Our system consists of a number of ECUs which are computational units in charge of specific control functions, such as engine control, suspension control, and driver assistance. Such ECUs are connected with each other through CAN buses [11]. To provide safety guarantees, the software components running on these ECUs have to exhibit deterministic performance through which the system strictly adheres to a set of predefined timing constraints. Therefore, AutoSec leverages the AUTOSAR (stands for automotive open system architecture) layered architecture for providing real-time capabilities together with separating the application layer from the hardware layer [7]. In AutoSec, we assume that the application software is composed of a number of periodic and preemptable tasks where each task consists of a number of atomic runnable entities. Moreover, we assume that the various tasks are fetched for execution according to a static schedule defined by a priority-driven scheduling algorithm.

Figure 2 demonstrates the architecture of our proposed AutoSec method. At the outset, AutoSec devises a tracing algorithm to collect the various timing parameters of the legitimate code. The collected traces are then used as an input to a clustering algorithm which generates a number of clusters, thus forming the timing model. To reduce the complexity of clustering the collected dataset, AutoSec exploits the principal component analysis (PCA) algorithm to reduce the dimensions without harming the detection accuracy (cf. Section V). During the runtime, the timing model is used as an input to a decision making algorithm which decides whether the examined executions behave abnormally. To this end, AutoSec is frequently activated to check the distance between each execution of the monitored task and the generated clusters.

Figure 3 illustrates an example of such a timing model where it visualizes the various clusters. In this example, the

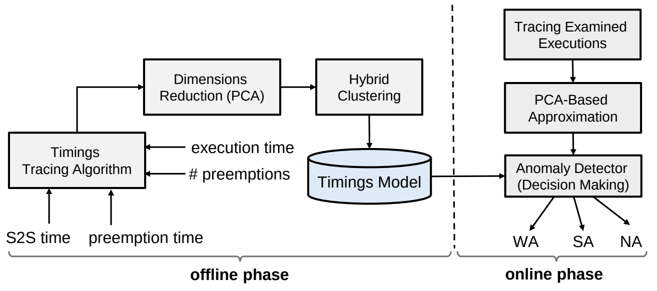


Fig. 2: System architecture of AutoSec showing the offline and online phases

executions—marked in black dots—located inside the boundaries of a certain cluster represents a class of legitimate code with normal behavior. Whereas, the black dots—surrounded by red circles in Figure 3—represents the abnormal executions which have been annotated as anomalies. Depending on the distance, AutoSec classifies the examined execution as either a weak anomaly (WA), a strong anomaly (SA), or a non-anomaly (NA) point. In the subsequent sections, we elaborate on the timing parameters and their tracing algorithm, before delving into the hybrid clustering method and the online algorithm.

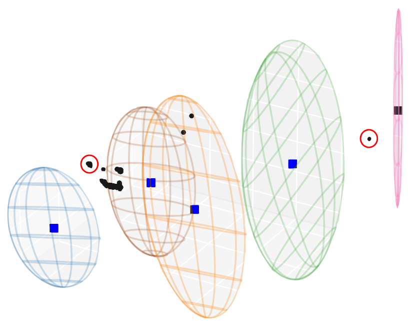


Fig. 3: Clusters describing the timing model where the black points representing the examined executions and the blue squares denote the middle point of each cluster,

B. Timing Parameters

In general, it is necessary to understand the anomalistic behavior which has to be detected via adopting AutoSec. As explained in Section II, considering solely the computation time of the various tasks while detecting anomalies is broadly not effective. Therefore, AutoSec relies on four different timing parameters, including the computation time, the S2S time, the preemption time, and the number of preemption. Figure 4 demonstrates the execution of two instances of task S_1 where the first instance J_{11} has been preempted twice by J_{21} and J_{31}. In this case, the S2S time represents the time elapsed between the start of execution of two successive instances. In fact, tracing the S2S time is crucial to detect whether a task skipped its execution or even executed more often than it was planned in the schedule. In addition, it implicitly indicates whether the previous instances were tampered with. For instance, if an attacker gained access to an ECU within a vehicle to deactivate a certain task, the system typically fetches the next lower-priority task for execution. In this case, the short S2S time between the instances of the lower-priority task has to be detected by the system to carry out a recovery mechanism.

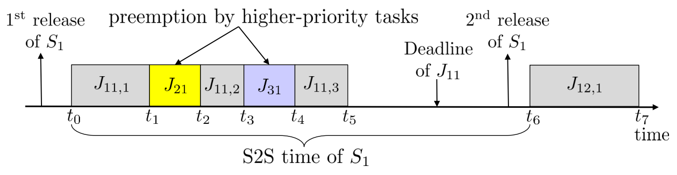


Fig. 4: Various timing parameters of a typical task

Through observing the number of preemptions, it is straightforward to detect the higher-priority tasks executed outside their planned time. Moreover, it enables detecting the unplanned executions. For instance, if a message-flooding attack has been performed through sending several CAN messages to a certain ECU, the corresponding service routine will preempt a currently-running task more often than expected. Accordingly, the system has to accurately detect the excessive number of preemptions to sidestep the severe consequences of such an attack. In addition to the number of preemptions, it is necessary to monitor the preemption time. For instance, an attacker may perform a message request using a non-existing message ID, thus forcing the corresponding control unit to run extensive computations to process this request. Indeed, such an attack leads to preempting a currently-running task for longer time than usual preemptions.

As an example, Figure 5 depicts the execution of the consecutive instances of a periodic task whose period is set to 10ms. Such a task is composed of five runnable entities $s_1, s_2, \cdots, s_5$. The normal behavior of these runnable entities is typically to execute consecutively every 10ms (cf. the left side of Figure 5). Nevertheless, the runnable s_4 has frequently been blocked due to long preemption during its second instance (cf. the right side of Figure 5). As a result, the second instance of the runnable s_5 has been postponed, thus resulting in a deviation of the S2S time. In this case, an efficient anomaly detection algorithm has to report such an abnormal behavior occurred due to the relatively long S2S time. In this paper, we assume no prior knowledge about the type of anomaly. Therefore, AutoSec treats the timing parameters evenly, i.e. all parameters have the same priority.

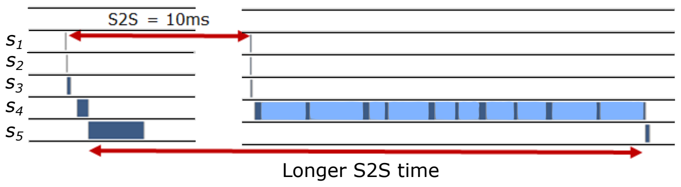


Fig. 5: Anomalistic behavior due to long preemptions where dark blue denotes execution time while light blue represents preemption time

IV. Timing Model Generation

Before delving into the process of generating the timing model, we explain the tracing algorithm developed for collecting the various timing parameters (cf. Figure 6). Such

```
Require: entity set $\mathcal{S} = \{s_1, \cdots, s_v\}$
 1: for all entity $s_i \in \mathcal{S}$ do
 2:     if curMP($s_i$) == None or newMP($s_j$) == None then
 3:         timestamp ← GetMCUTime()
 4:         newMP ← debugger(timestamp)
 5:         Start(newMP, timestamp)                    ▷ S2S time
 6:         timestamp ← GetMCUTime()
 7:         Finish(newMP, timestamp)          ▷ computation time
 8:     else if newMP($s_j$).status == running then
 9:         timestamp ← GetMCUTime()
10:         Preempt(curMP, newMP, timestamp) ▷ $N_{preempt}$
11:         timestamp ← GetMCUTime()
12:         Continue(curMP, newMP, timestamp) ▷ $T_{preempt}$
```

Fig. 6: Tracing the four timing parameters

an algorithm is granularity-agnostic where it can be adopted to trace the timing parameters of the tasks or the runnable entities. Therefore, we refer to the software component being executed as an entity s_i. For this algorithm, we have v entities where each entity s_i can be preempted by a higher-priority entity s_j. The core idea behind our algorithm is to enable the debugger to collect various timing information at a set of predefined measuring points. Such measuring points occur in the following cases: (1) the start of execution of the entity s_i, (2) the preemptions by higher-priority entities, e.g. s_j, and (3) the end of execution of the entity s_i.

The algorithm mainly relies on four functions, namely *start*, *continue*, *preempt*, and *finish*. As Figure 6 depicts, the *start* function is executed when a measuring point $curMP$ for the currently-running entity s_i or a new measuring point $newMP$ for a preempting entity s_j is to be defined (cf. line 2). Once a new measuring point is defined, the *start* function stores the timestamp, before setting the other parameters, i.e. computation time, number of preemption, and preemption time, to zero. Afterwards, the *start* function records the timestamp of starting the next instance. The difference between these two values represent the S2S time of the currently-running instance. The *preempt* function is typically executed when the current measuring point of s_i is preempted by an entity s_j whose measuring point $newMP(s_j)$ was previously defined. once activated, the *preempt* function first estimates the computation time of the executed part of the preempted task s_i. In addition, it increments the number of preemptions, before recording the current timestamp. Upon resuming the execution of the entity s_i, the *continue* function is executed where it records the new timestamp to estimate the preemption time. Finally, the *finish* function computes the execution time of s_i, before activating the inference algorithm of AutoSec. Below, we elaborate on the process of clustering the collected timing parameters.

A. Clustering Requirements

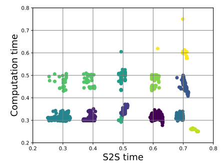
To generate the timing model, we have to employ a clustering algorithm to define the different groups of executions. However, there exist plenty of clustering algorithms. To select the most well-suited algorithm for our problem, we defined six requirements which have to satisfied by the selected clustering algorithm, including: (**R1**) supporting unsupervised learning where the abnormal behavior may occur even during the normal execution of the legitimate tasks, (**R2**) requiring no prior knowledge about the number of clusters, (**R3**) dealing with unbalanced dataset in which the points are not evenly distributed between the different clusters, (**R4**) supporting non-linearly separable dataset, (**R5**) dealing with noisy dataset, and finally (**R6**) reducing the training time necessary to generate the timing mode together with enabling parameters tuning.

While developing AutoSec, we examined six widely-used clustering algorithms, including the K-Means algorithm [9], the spectral clustering algorithm [12], the affinity propagation (AP) algorithm [13], the Gaussian mixture models (GMM) [14], and the density-based spatial clustering of applications with noise (DBSCAN) [8]. The principle of operation of such algorithms is left beyond the scope of this paper. However, it is necessary to mention that some algorithms, e.g. K-Means, take the number of clusters as an input. For these algorithms, we adopted the Silhouette method [10] for obtaining a range of potential number of clusters. From this range, we select the value which results in the minimum average distance between the points in each cluster.

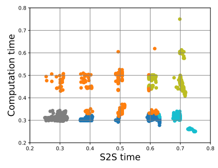
Additionally, the DBSCAN algorithm generally relies on grouping data points close to each other based on a distance measure and a minimum number of points in each cluster. Furthermore, DBSCAN marks the data points existing in the low-density regions as outliers. To this end, DBSCAN requires prior knowledge about the minimum number of points in each cluster $MinSamples$, the minimum neighboring points $MinNeighbors$, and the reachability distance ϵ—defined as the distance between a point of interest and a seed point. Indeed, finding well-suited values for these parameters requires achieving a reasonable compromise between the number of clusters and the points sparsity in each cluster. Based on our experiments, we found that setting $MinSamples$ to $MinNeighbors$ to three and five, respectively, broadly achieves this requirement. To determine the distance ϵ, we adopt a *semi-automatized* technique which relies on expert knowledge for selecting a value from a range defined by the k-Distance graph method [8].

B. Clustering Algorithms

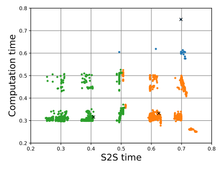
In this section, we assess the performance of the aforementioned clustering algorithms for the sake of selecting the one which satisfies our defined requirements. It is worth mentioning that the Hopkins statistic H [15] has initially been used to measure the cluster tendency of our dataset. Such a metric denotes the probability that a given dataset is generated by a uniform data distribution, i.e. testing the spatial randomness of the data. We found that the Hopkins value H is always above 60%, which implies that our dataset can be feasibly clustered. Figure 7 demonstrates a comparative study between the various clustering algorithms in the 2D feature space, i.e. S2S time versus computation time. Figure 7a depicts the clusters generated by the K-Means algorithm. Obviously, K-Means does not offer valid clusters where the yellow dots—located on the upper right side of the figure—have joined the yellow cluster despite being relatively distant from the center of the yellow cluster. Such a result of the K-Means algorithm occurs owing to the imbalanced point density defined in terms of the number of points per cluster. Figure 7b demonstrates the clusters generated by the spectral clustering algorithm. Clearly, the spectral clustering has a similar performance as the K-Means algorithm where it is broadly sensitive to the imbalanced density of the dataset.



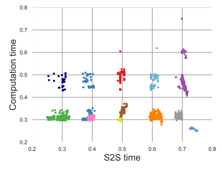
(a) K-Means



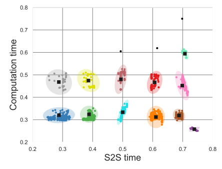
(b) Spectral clustering



(c) Affinity propagation



(d) Gaussian mixture



(e) DBSCAN

Fig. 7: Comparative study of six clustering algorithms where each cluster is identified by its color

Along a similar line, the AP algorithm produces solely three clusters (cf. Figure 7c), which is not consistent with the number of clusters obtained by the Silhouette method. Furthermore, we found that some points were misplaced due to computing the centroid of each cluster (marked x in Figure 7c) as the mean value of the data points. In case of unbalanced datasets, considering the mean value as the centroid may result in obtaining biased center points. Therefore, AP was not decisive about the belonging of the middle points. Figure 7d depicts the clusters generated by the GMM algorithm which requires prior knowledge about the number of clusters in the dataset. Obviously, the GMM algorithm is sensitive to the points density, e.g. the yellow cluster is made of points with relatively high dispersal.

Finally, Figure 7e demonstrates the clusters generated by the DBSCAN algorithm. As the figure shows, DBSCAN managed to identify the outliers while exploiting the points density to properly define the various clusters. Table I depicts the ability of each clustering algorithm to satisfy the defined requirements. The table shows that DBSCAN is the algorithm which satisfies all the requirements where it has the smallest training time, requires no prior knowledge about the number of clusters, and is insensitive to the noise and the imbalanced density. Therefore, we adopted the DBSCAN algorithm while designing our AutoSec approach.

C. Hybrid Clustering Method

After opting for DBSCAN, we observed that it sometimes combines several clusters when there exist bridge points between these clusters. Figure 8 demonstrates an example of clusters generated for a subset of the training dataset. In this

TABLE I: Requirements evaluation of different clustering methods using a dataset of 3183 instances of execution

	Requirements					
Classifiers	R1: Unsupervised Learning	R2: Unknown Number of Clusters	R3: Unbalanced Clusters	R4: Non-linearly Separable Clusters	R5: Detected Outliers	R6: Training Time
K-Means	✓		✓	✓		N.A.
Spectral Clustering	✓		✓	✓		1.71 Sec
Affinity Propagation	✓	✓	✓	✓		3.12 Sec
Gaussian Mixture	✓		✓	✓		0.17 Sec
DBSCAN	✓	✓	✓	✓	✓	0.14 Sec

figure, each cluster is described using boundaries embodying its data points. However, some special clusters, e.g. the red and blue clusters, are characterized using two distinct boundaries i.e. two sub-clusters. Clearly, each special cluster typically consists of two dense regions which have not been recognized as a single cluster thanks to the data points forming a bridge between these two neighboring regions. In fact, AutoSec leverages the distance between an examined point s_i and the midpoint of a cluster s_0 to decide whether the examined point s_i represents an anomaly. Therefore, it is highly crucial to determine a fine-grained description of the clusters for the sake of precisely detecting the anomalistic executions.

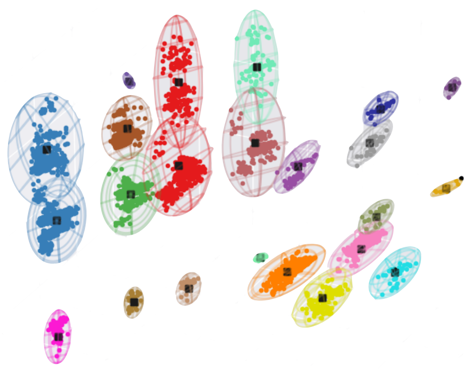


Fig. 8: Sub-clusters (e.g. red and blue clusters) existence due to the bridge points

In this context, AutoSec adopts a hybrid clustering algorithm which leverages DBSCAN as a pre-processing tool to remove possible outliers together with generating the initial clusters. Figure 9 depicts the hybrid clustering method for obtaining a fine-grained description of the clusters. In AutoSec, we iterate over each cluster generated by DBSCAN to determine the sub-clusters, if any. To this end, we adopt the Silhouette method in combination with the K-Means method. Specifically, the K-Means method is iteratively executed on each cluster with different values of K, i.e. the number of possible sub-clusters. In each iteration, the Silhouette index is utilized to examine the closeness of each point to the points in the neighboring sub-clusters. If the Silhouette index, for any value of K, surpasses the predefined threshold, the corresponding value of K will be taken into the further use. Based on our evaluations, the Silhouette threshold β and the maximum number of sub-clusters are set to 70% (Hopkins value) and four, for an accurate estimation of the sub-clusters.

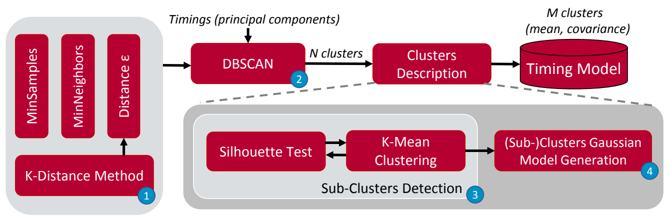


Fig. 9: Clustering procedure including sub-clusters estimation

A primary limitation of the DBSCAN and K-Means methods is the lack of uncertainty measure or probability that tells us how much a new data point is associated with a specific cluster. To overcome this limitation, the detected (sub-)clusters have been described as a set of Gaussian models. Accordingly, new data points can be added to the closest cluster based on probabilities generated by the relevant Gaussian model. Finally, the obtained models of each (sub-)cluster, i.e. midpoint and covariance, are utilized to generate a data description file compatible with the ECU software. Below, we explain the different steps of the AutoSec algorithm executed during the runtime for checking the behavior of the examined executions.

V. AutoSec Online Algorithm

Figure 10 demonstrates the various steps of our proposed AutoSec algorithm. Such an algorithm takes as inputs the timing parameters $\{T_{s2s}, e_i, N_{preempt}, T_{preempt}\}$ of a certain entity s_i together with a set of clusters $\mathcal{C}$ where each cluster c_i may comprise a number of sub-clusters $z_1, \cdots, z_w$. In fact, we noticed that the number of preemptions $N_{preempt}$ and their duration $T_{preempt}$ become broadly correlated in case of normal execution (cf. Figure 11). Based on this observation, AutoSec leverages the PCA algorithm [16] to create new dimensions, referred to as the principal components $P_1, \cdots, P_w$, where each principal component represents a combination of all original dimensions. Afterward, AutoSec selects, out of these principal components, the most important ones for detecting the misbehaved executions (cf. line 2). Thus, PCA reduces the feature space while preserving the important information embedded in the dataset. As a result, the clustering time can be shortened—through reducing the Euclidean distance computations—without negatively impacting the detection accuracy.

After generating the principal components $s_{pca} = \{P_1, P_2, P_3\}$, they have to be transformed into the local coordinate system x, y, z of the clusters (cf. line 5). To

```
Require: clusters set $\mathcal{C} = \{c_1, \cdots, c_u\}$, test points $\mathcal{S} = \{s_i, \cdots, s_v\}$ where $s_i = T_{S2S}, e_i, N_{preempt}, T_{preempt}$
1: for all test point $s_i \in \mathcal{S}$ do
2:     $s_{pca} \leftarrow$ PCA($s_i$)                ▷ dimensions reduction
3:     for all cluster $c_i \in \mathcal{C}$ do
4:         for all sub-cluster $z_i \in c_i$ do
5:             $s_{local} \leftarrow$ transform($s_{pca}, z_i$)
6:             $\theta_{local}, \phi_{local} \leftarrow$ findAngles($s_{local}, z_i$)
7:             $d_{local} \leftarrow$ findDistance($s_{local}, z_i$)
8:             $\rho_{local} \leftarrow$ findRadius($s_{local}, z_i, \theta_{local}, \phi_{local}$)
9:             if the ratio $d_{local}/\rho_{local} > 1$ then
10:                $R \leftarrow$ Append($d_{local}/\rho_{local}$)
11:    $R_{min} \leftarrow$ minimum($R$)
12:    if $R_{min} \geq \alpha_{strong}$ then
13:        $S_{strong} \leftarrow s_i$            ▷ strong anomaly detection
14:    else if $R_{min} \geq \alpha_{weak}$ & $R_{min} < \alpha_{strong}$ then
15:        $S_{weak} \leftarrow s_i$              ▷ weak anomaly detection
```

Fig. 10: AutoSec algorithm

quantify how close is an examined point to a certain (sub-)cluster, AutoSec normalizes the distance d_{local} using the radius of each (sub-)cluster ρ_{local} in the direction of the examined point x_i, y_i, z_i. To this end, the algorithm iterates over all (sub-)clusters to compute the Euclidean distance d_{local} and angles $\theta_{local}, \phi_{local}$ between s_i and the various (sub-)clusters. Equation 1 defines the distance d_{local} in terms of the coordinates of the midpoint x_0, y_0, z_0 for each (sub-)cluster. Whereas, Equation 2 estimates the radius of each (sub-)cluster in the direction of the examined point in terms of three radii a, b, c—which define the ellipsoid of the (sub-)cluster—together with its spherical angular $\theta_{local}, \phi_{local}$ (cf. lines 7 and 8). If the normalized distance $\frac{d_{local}}{\rho_{local}}$ does not exceed one, AutoSec considers the examined point s_i as a legitimate component executed without anomalistic behavior since it lies within one of the pre-defined clusters. After iterating over all (sub-)clusters, AutoSec checks the minimum normalized distance R_{min} representing the distance to the closest (sub-)cluster. If the distance R_{min} exceeds an upper threshold α_{strong}, the examined point s_i is classified as a strong anomaly (cf. line 12). However, it is considered as a weak anomaly whenever the distance R_{min} lies in the predefined range $[\alpha_{weak}, \alpha_{strong}]$ (cf. line 14). Such classification is crucial to enable the system from making the optimal recovery mechanism, e.g. reseting the control unit or warning the driver.

$$d_{local} = \sqrt{(x_i - x_0)^2 + (y_i - y_0)^2 + (z_i - z_0)^2} \tag{1}$$

$$\rho = \frac{abc}{\sqrt{(ab)^2 \cos^2 \phi + (ac)^2 (\cos\theta \sin\phi)^2 + (bc)^2 (\cos\theta \sin\phi)^2}} \tag{2}$$

VI. Performance Evaluation

To show the effectiveness of AutoSec, we tested our system in a pseudo real-world scenario. It is worth noting that the scenario comprises a simplified simulation setup for demonstration purposes and not a real sub-system of a vehicle. We first describe the setup of our evaluation, before we discuss the evaluation results.

A. Experimental Setup

To test our proof-of-concept implementation of AutoSec, we designed a testbed composed of a real ECU, i.e. an AURIX three-core board running at 200MHz, which communicates through a CAN bus with two *virtual* ECUs. In this simulated setup, the virtual ECUs are responsible for controlling a simple vehicle's engine while the real ECU is dedicated to control the brakes. In our scenario, the input signals coming from a brake pedal are initially preprocessed by a virtual ECU, before forwarding the output to the real ECU as a set of CAN messages. Afterwards, the real ECU estimates the braking force, before sending it back to the corresponding virtual ECU. In our evaluations, we are mainly interested in tracing the timing parameters of the software executed on the real ECU, while the two virtual nodes are used as communication nodes.

Practically speaking, the CANoe simulation tool [17] has been used to create the virtual ECUs together with analyzing their communication with the real ECU. In addition, a VN5610 Ethernet/CAN interface and an iSystem iC5000 debugger have been connected to an Intel Xeon machine running at 2.9GHz and equipped with 64GB of RAM. The iSystem debugger is mainly used for porting our code into the real ECU, debugging and tracing the timing parameters. The VN5610 Ethernet/CAN interface has been utilized as a communication mean between the real and virtual ECUs. It is worth mentioning that the MICROSAR software packages [18] have been used to implement the requirements and specifications of AUTOSAR.

In general, it is significantly crucial in automotive applications to minimize the false positives, i.e. false alarms, while achieving a reasonable number of true positives. The intuition is to sidestep warning the driver with potential security breaches unless the anomaly detection system is highly certain that a breach took place. Similarly, it is indeed required to minimize the false negatives, i.e. missed detections, to capture all breaches before destabilizing the control decisions. In this context, AutoSec is evaluated in terms of three accuracy measures, including the *precision*, the *recall*, and the *false positive rate* (FPR). The precision $P = \frac{TP}{TP+FP}$ denotes the fraction of true anomalies out of all detected cases, where TP denotes the true positives and FP is the false positives. For example, a precision = 95% implies that out of 100 detected cases (i.e. outliers), 95 are true anomalies, whilst the other five cases represent normal behavior. The second metric, i.e. recall $R = \frac{TP}{TP+FN}$ denotes the fraction of detected anomalies out of all injected anomalies, where FN represents the false negatives. Finally, the FPR metric represents the fraction of instances of normal behavior surpassed the adopted threshold and have been annotated as anomalies, i.e. $FPR = \frac{FP}{FP+TN}$, where TN denotes the true negatives.

In fact, the process of evaluating anomaly detection algorithms is predominantly not straightforward owing to the lack of ground-truth data. As a workaround, we opted for creating our ground-truth through carrying out the following three steps: (1) adopting AutoSec to a set of collected timing traces where the sensitivity thresholds α_{weak} and α_{strong} are adjusted so that no anomalies being detected in this dataset, (2) deliberately injecting a set of different anomalies, and finally (3) adopting AutoSec to determine its ability in detecting the injected anomalies. Specifically, the injected malicious code has been designed to tamper with the S2S time and the computation time. For instance, an attacker can inject a malicious code to partially or completely bypass the execution of certain runnables, thus reducing their computation time. In our scenario, the injected code performs random computations whose length increases over time, before writing the output to the memory. The intuition behind this design is to evaluate the sensitivity of AutoSec to (1) the variation in the computation time and (2) the existence of memory dependencies. It is worth mentioning that the preemption anomalies were hard to be examined in our evaluations due to the high uncertainty of external triggers with high priorities. Table II summarizes the parameters and their values which have been used throughout the evaluations.

TABLE II: parameters used in the evaluations

System Parameter	Value	System Parameter	Value
trace length	300 s	sensor frequency	5 ms
anomalies frequency	1/50	S2S lower bound	8.5 ms
S2S upper bound	11.5 ms	clustering threshold β	70%
upper threshold α_{strong}	4	lower threshold α_{weak}	2.5

B. Detection Accuracy

In this section, we evaluate the ability of AutoSec in detecting the various injected anomalies. We describe the results obtained while tracing the timing parameters on the task level as well as on the runnable level. It is worth mentioning that the anomalies $A_s^{(1)}, \cdots, A_s^{(N)}$ are injected every 50th instance of the executed task. In this scenario, the length of such anomalies L_s is deliberately increased every ten consecutive anomalies, e.g. $L_s^{(11)} = (1 + \delta) \times L_s^{(1)}$, where δ denotes the deviation factor. Figure 11 demonstrates the four timing parameters for different instances of execution in the runnable level. In this figure, the red dots represent a set of injected computation time anomalies. Obviously, the S2S times of such executions lie within the normal ranges, i.e. between the S2S upper and lower bounds (cf. Table II). Nevertheless, their computation times are relatively long, i.e. circa 110 μs. In fact, long computation times in the range of 110 μs typically occur when the executed task is frequently preempted.

Through observing the third and fourth timelines, i.e. the number of preemptions and the accumulated preemption time, we found that these red executions were not interrupted, i.e. number of preemptions is equal to zero. Hence, there exists no valid reason for the red executions to have a relatively long computation time. In these cases, AutoSec annotates these malicious executions as possible anomalies. Table III summarizes the detection accuracy results where the shown recall, precision, and FPR values represent average readings over six test cases for the runnable level and three test cases for the task level. In this table, the third row shows the deviation factor δ of the subsequent anomalies relative to the length of the initial anomaly, i.e. $L_S^{(1)}$. At small values of the deviation factor δ, the timing parameters of the injected anomalies become relatively close to the normal executions, thus making it challenging to detect those anomalies.

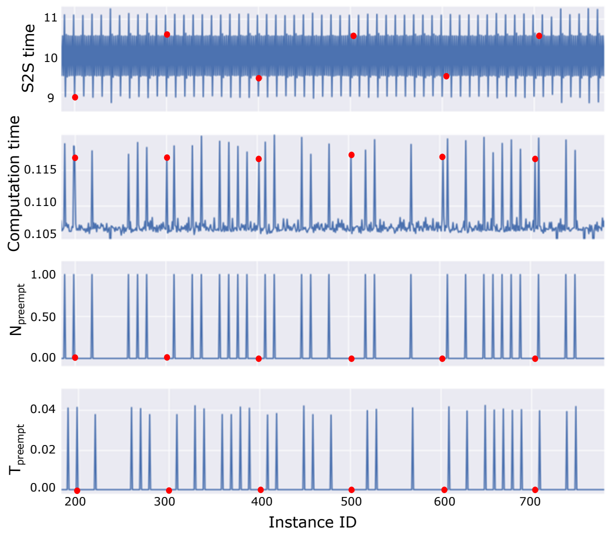


Fig. 11: Traced timing parameters with marking the computation time anomalies in red

For the S2S time anomalies, AutoSec has extremely few false positives where the precision metric approaches 100% and FPR approaches 0.01%, thus drastically reducing the number of false alarms forwarded to the system/driver. Additionally, the evaluations show that AutoSec achieves less false negatives (at least by 47%) in both levels when the deviation factor is increased. For the computation time anomalies with small values of the deviation factor, AutoSec achieves less false positives (on average by 39%) in the fine-grained runnable level compared to the coarse-grained task level. However as the deviation factor is increased, AutoSec produces approximately the same amount of false positives in both granularity levels. As expected, increasing the deviation factor within the runnable level leads to further reducing the FPR rate (at most by 50%) while having no false negatives, i.e. R = 100%. Accordingly, we can conclude that AutoSec offers a reasonable detection accuracy even with small values of the deviation factor δ.

To further understand the behavior of AutoSec, Figures 12a–12f visualize another set of experiments in which several tests have been run for different values of the deviation factor δ. The figures depict the recall and precision results on both granularity levels. In the scatter plots, each point represents the average value of the results obtained from repeating the test ten times. At the outset, Figure 12a demonstrates the recall on the runnable level (i.e. R-level) in case of the S2S time and the computation time anomalies. Obviously, AutoSec achieves higher recall (on average by 55%) while reacting to the computation time anomalies. Accordingly, AutoSec detects the

TABLE III: Detection accuracy in case of the S2S time and the computation time anomalies

	S2S Time Anomalies				Computation Time Anomalies			
	Runnable Level		Task Level		Runnable Level		Task Level	
Deviation δ	[1..25%]	[25..40%]	[1..25%]	[25..40%]	[1..10%]	[5..15%]	[1..10%]	[5..15%]
FPR	0.00605632%	0.02004812%	0.01976089%	0.01978351%	0.030305%	0.015%	0.0499%	0.010068%
Recall	33%	79%	34%	64.52%	76.5%	100%	22%	56.5%
Precision	99%	98.77%	97.18%	98.46%	98.115%	99.26%	98.81%	99.09%

computation time anomalies more efficiently than the S2S time in the R-level. Such a result predominantly occurs since the computation time anomalies—in our settings—directly affect the S2S time of the periodic runnable entities. Whereas, the S2S time anomalies may have no impact on the computation time of these entities. Figure 12b shows that AutoSec achieves on the task level, i.e. T-level, comparable recall results while reacting to both anomaly types. Figure 12c demonstrates the recall distribution on both granularity levels. As it can be seen in the figure, AutoSec on the R-level has higher recall (on average by 28%) compared to the T-level. The main reason behind this result is the coarse resolution of the traced timings on the T-level. This implies that small deviations in the range of few microseconds within one runnable entity traced under the T-level mostly will not be detected. Such deviations can be easily detected if that runnable entity is monitored separately, i.e. tracing on the R-level.

Similarly, Figure 12d depicts the precision achieved by AutoSec while detecting the S2S time and computation time anomalies on the R-level. Clearly, AutoSec has an extremely small number of false positives while reacting to both anomaly types. Such a result has also been obtained while tracing the timing parameters on the task level, as it can be seen in Figure 12e. Such a result confirms the ability of AutoSec to efficiently differentiate between the legitimate and the malicious executions thanks to considering a multi-dimensional feature space. Figure 12f shows the precision distribution of AutoSec on the two granularity levels. Despite having a relatively high variability, the precision on the R-level is mostly concentrated in the range between 98.5% and 100%. Hence, AutoSec produces extremely few false positives and false negatives while reacting to the S2S time and computation time anomalies.

Table IV illustrates another set of experiments whose goal is to provide a comparative study between AutoSec in the R-level and two baseline methods, including SecureCore [6] and LogSed [4]. In this experiment, two different runnable entities have been monitored, including (1) an aperiodic entity and (2) a periodic entity with 10 ms period. Each test has been run for two minutes and then repeated ten times where the obtained results are averaged. Clearly, AutoSec achieves higher recall on average by 26% and 8% relative to SecureCore and LogSed, respectively. Thus, AutoSec has higher ability to reduce the false alarms sent to the system/driver. In addition, AutoSec has much smaller false positives compared to the two baseline methods. Along a similar line, AutoSec achieves higher precision (on average by 21%) than LogSed while having approximately similar performance as SecureCore. Nevertheless, these results—achieved by AutoSec—still have to be further improved to drastically reduce the probability of triggering false alarms in order to meet the strict requirements in the automotive industry.

TABLE IV: Comparing AutoSec and the baseline methods

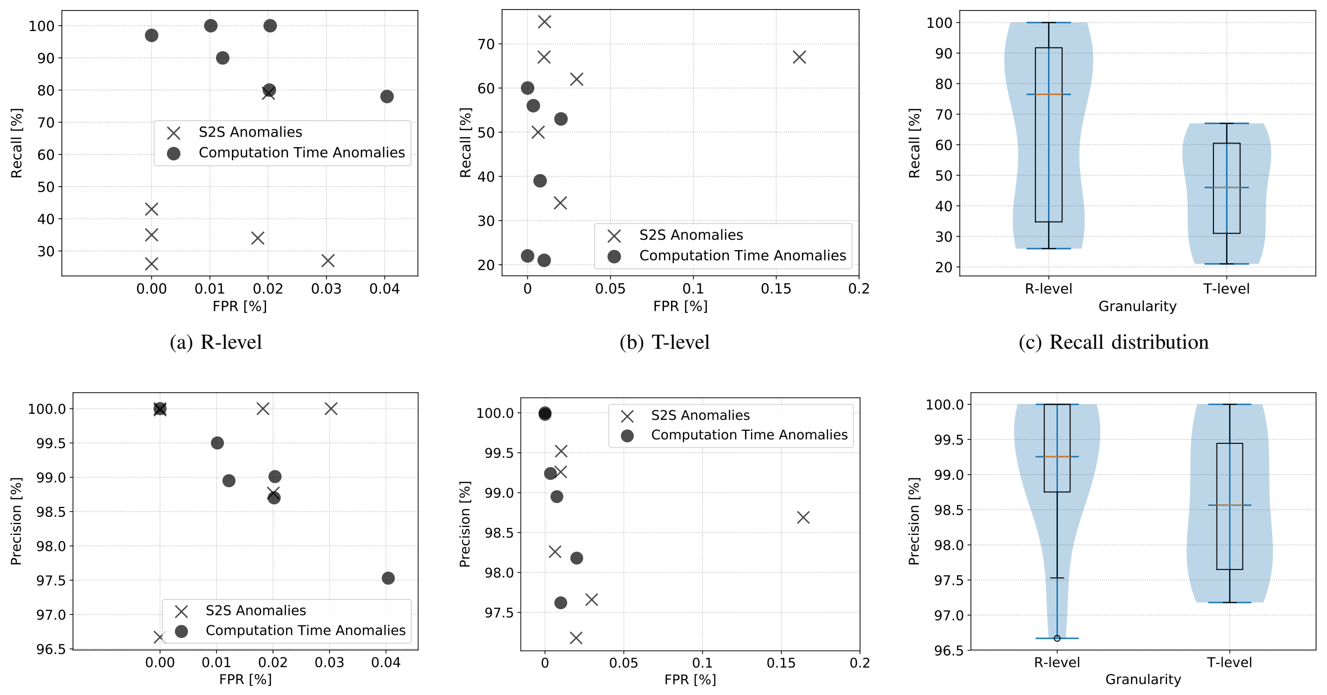
Methods	FPR	Recall	Precision
SecureCore	0.39385%	66.775%	99.215%
LogSed	1.12500%	82.500%	78.500%
AutoSec	0.00750%	90.000%	98.950%

C. *Overhead of AutoSec*

In this section, we discuss the overhead of executing our AutoSec approach. Although the processing overhead significantly depends on the hardware under which AutoSec is running, an insight can still be gained on the overhead from observing the processing time required for getting the timestamps together with performing the inference. In our testbed, the speed of the adopted control unit is 200 MHz where AutoSec has been executed on a single core out of three available cores. Under these settings, the processing time of our implementation can be simply measured using the internal clock of the control unit. We found that AutoSec requires between 40 and 200 μs for processing a single runnable. Specifically, the processing time depends on the number of clusters where longer processing time is mostly needed as the number of available clusters is increased. Furthermore, the processing time also depends on whether the examined runnable entity represents an actual anomaly. If the tested runnable entity behaves in an malicious manner, AutoSec has to measure its distance, in the feature space, to each cluster. Such computations involve multiple multiplications, divisions and using square root functions. Otherwise, the distance measurements are avoided if the entity resides in one of the predefined clusters. Accordingly, executing AutoSec on the runnable level may incur relatively high overhead compared to the T-level tracing. Hence, it is highly recommended to perform T-level tracing to reduce such an overhead while achieving a reasonable detection accuracy.

VII. Conclusion & Future Work

In this paper, we presented AutoSec, a host-based AUTOSAR-compliant anomaly detection algorithm. AutoSec



(a) R-level (b) T-level (c) Recall distribution

(d) R-level (e) T-level (f) Precision distribution

Fig. 12: Evaluating the detection accuracy on both granularity levels

employs four timing parameters to observe the behavior of the software executions. In this context, the abnormal executions are detected through comparing them with a timing model generated during design time. To generate such a model, AutoSec introduces a hybrid clustering algorithm which generates a fine-grained representation of the timing parameters collected from a set of legitimate executions. As a proof-of-concept, we created a testbed composed of three ECUs which communicate with each other through a CAN bus. The evaluation results showed that AutoSec is highly effective in reducing the false positives and the false negatives. Additionally, the results indicated that executing AutoSec on the R-level achieves better recall and precision at the expense of incurring additional overhead due to the fine-grained tracing. A logical extension of this work involves examining AutoSec on the adaptive AUTOSAR layered architecture where dynamic scheduling strategies are employed. Finally, we plan to investigate the integration of more timing parameters, e.g. arrival time, blocking time, and completion time, while generating the timing model.

References

[1] J. Wang, J. Liu, and N. Kato, "Networking and communications in autonomous driving: A survey," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1243–1274, 2018.

[2] X. Yang, L. Liu, N. H. Vaidya, and F. Zhao, "A vehicle-to-vehicle communication protocol for cooperative collision warning," in *MOBIQUITOUS*. IEEE, 2004, pp. 114–123.

[3] C. Miller and C. Valasek, "Remote exploitation of an unaltered passenger vehicle," *Black Hat USA*, vol. 2015, p. 91, 2015.

[4] T. Jia, L. Yang, P. Chen, Y. Li, F. Meng, and J. Xu, "LogSed: Anomaly diagnosis through mining time-weighted control flow graph in logs," in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*. IEEE, jun 2017.

[5] M. F. B. Abbas, A. Prakash, and T. Srikanthan, "Power profile based runtime anomaly detection," in *2017 TRON Symposium (TRONSHOW)*. IEEE, 2017, pp. 1–9.

[6] M.-K. Yoon, S. Mohan, J. Choi, J.-E. Kim, and L. Sha, "SecureCore: A multicore-based intrusion detection architecture for real-time embedded systems," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, apr 2013.

[7] S. Fürst and M. Bechter, "Autosar for connected and autonomous vehicles: The autosar adaptive platform," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*. IEEE, 2016, pp. 215–217.

[8] M. Ester, H.-P. Kriegel, J. Sander, X. Xu *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Kdd*, vol. 96, no. 34, 1996, pp. 226–231.

[9] M. David, "Chapter 20. an example inference task: Clustering," *Information Theory, Inference and Learning Algorithm*, 2003.

[10] R. C. de Amorim and C. Hennig, "Recovering the number of clusters in data sets with noise features using feature rescaling factors," *Information Sciences*, vol. 324, pp. 126–145, 2015.

[11] S. Fürst, "Challenges in the design of automotive software," in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2010, pp. 256–258.

[12] U. Von Luxburg, "A tutorial on spectral clustering," *Statistics and computing*, vol. 17, no. 4, pp. 395–416, 2007.

[13] B. J. Frey and D. Dueck, "Clustering by passing messages between data points," *science*, vol. 315, no. 5814, pp. 972–976, 2007.

[14] S. Jafatnejad, G. Castignani, and T. Engel, "Revisiting gaussian mixture models for driver identification," in *2018 IEEE International Conference on Vehicular Electronics and Safety (ICVES)*. IEEE, 2018, pp. 1–7.

[15] A. Banerjee and R. N. Dave, "Validating clusters using the hopkins statistic," in *2004 IEEE International conference on fuzzy systems (IEEE Cat. No. 04CH37542)*, vol. 1. IEEE, 2004, pp. 149–153.

[16] I. Jolliffe, *Principal Component Analysis*. Wiley Online Library, 2002.

[17] "CANoe 8.2: ECU & Network Testing on Highest Level," Vector Informatik GmbH, online; accessed April-2019. [Online]. Available: http://vector.com/vi_canoe_en.html

[18] "MICROSAR — The Smart Implementation of the AUTOSAR Classic Standard," Vector Informatik GmbH, online; accessed April-2019. [Online]. Available: https://www.vector.com/de/en/products/products-a-z/embedded-components/microsar/