# On-board Diagnosis: A First Step from Detection to Prevention of Intrusions on Avionics Applications

Aliénor Damien, Pierre-François Gimenez, Nathalie Feyt, Vincent Nicomette, Mohamed Kaâniche, Eric Alata

# On-board Diagnosis: A First Step from Detection to Prevention of Intrusions on Avionics Applications

Aliénor Damien*, Pierre-François Gimenez†, Nathalie Feyt*, Vincent Nicomette†, Mohamed Kaâniche†, Eric Alata†
*Thales AVS, Toulouse, FRANCE, Email: {firstname}.{lastname}@fr.thalesgroup.com
†LAAS-CNRS, Université de Toulouse, CNRS, Email: {firstname}.{lastname}@laas.fr

***Abstract*—Nowadays, air travel is one of the safest transportation means. While safety is historically well integrated into avionics systems, it is becoming increasingly important to take into account the security of such systems for the future. In particular, Host-based Intrusion Detection Systems (HIDS) are commonly used in traditional information systems to improve their security. The adaptation of such systems for deployment inside an aircraft has been studied in another work and has shown to be effective in detecting anomalous behavior in an avionic application. However, the detection itself is not sufficient to provide an on-board reaction, and to prevent such intrusion. This paper proposes to improve such HIDS by introducing a signature-based system capable of providing a first diagnosis after the detection of an anomalous behavior. The proposed diagnosis approach is based on the definition of the signature of an alert, and its comparison with a knowledge database that is regularly updated throughout aircraft lifetime. This approach has been implemented on a real avionic computer and yielded good results in terms of classification accuracy and resources consumption.**

***Index Terms*—Intrusion Detection System, Security, Avionics, Embedded, Real-Time**

## I. Introduction

Major advances have been achieved over the last few decades to ensure aircraft passengers safety. Traditionally, aircraft systems design has mainly focused on taking into account failures and threats that have accidental causes. However, more recently, attention has been also focused on providing protection against potential malicious threats. In particular, to meet the growing need for connectivity and improve the passenger experience, aircraft systems are constantly evolving and integrating more and more connected services and devices. From a security point of view, this trend is leading to a larger attack surface. With the continued increase of threats targeting embedded systems, the potential malicious alteration of an aircraft application must be seriously considered for future systems. Among the various solutions to address such threats, Host-based Intrusion Detection Systems (HIDS) are widely used in information systems security.

However, traditional HIDS need to be adapted to the specific constraints and stringent requirements of embedded avionic computer systems and applications, in particular in the context of Integrated Modular Avionics (IMA) architectures. An IMA system is organized as a network of computing modules, each supporting several applications, possibly with mixed criticality levels. Space and time segregation mechanisms are used to run mixed-criticality software on the same module in compliance with the ARINC 653 standard. Each application is composed of one or more partitions. Each partition is statically and periodically assigned an execution slot, as well as memory resources protected by the underlying operating system.

In [1], the authors described the design and the implementation of a HIDS adapted to such embedded real-time and critical context, taking into account the following objectives: **O1** Preserving the real-time execution of the other functions, **O2** Proposing limited evolution for legacy aircraft, **O3** Performing real-time detection, **O4** Having a small memory footprint, **O5** Providing reliable and explainable results and **O6** Being efficient even on "black-box" applications.

The proposed HIDS implements an anomaly-based approach adapted to the IMA context and fulfilling the above objectives, based on the monitoring of ARINC 653 API calls. During aircraft integration phase, a model of the legitimate behavior of an avionic application is built based on data collected by monitoring specific features related to the application itself or to its environment. During the operational phase, alerts are raised when the behavior of the application exhibits significant deviations from this model. The implementation of such HIDS on a real avionic computer showed the relevance of such approach, and exhibited very good results in terms of detection efficiency and resource usage. However, this first version of the HIDS was designed to only detect anomalous behavior, without providing diagnosis about the alert raised. Such capability is an integral part of the overall HIDS approach proposed, but was not explored in the first implementation.

This paper explores how to provide a first diagnosis of the anomalous behavior detected, directly on-board. Such functionality is a first step towards a future automatic reaction on-board, and potential capability to block the attacks (and not only to detect them). In this case, the ability to provide reliable and explainable results is extremely important.

The diagnosis approach, proposed and implemented in this paper, relies on concepts similar to signature-based HIDS. A knowledge database is implemented onboard the aircraft to provide a diagnosis based on already-known anomalous behaviors (including attacks, safety-related failures, and even known false alerts). The main advantage of using a database is the ability to update it on a regular basis, so that the diagnosis of anomalies detected by the HIDS is as accurate as possible. It should be noted that in the avionics context, the most frequent update of a database is every 28 days, while the update of an

application code is extremely rare (in fact, avionic applications are not designed to be updated during the aircraft life cycle).

The anomaly-based detection approach implemented in the HIDS is based on the application code and should not be updated as it represents the normal behavior of the application, which is not expected to change over time, except in the unlikely event of an update of the application itself. Building a model of normal behavior is also much more complex than adding a known attack to a knowledge database. In other words, the detection capability of the HIDS cannot be improved on a regular basis because the applications embedded onboard are very seldom updated. On the other hand, the diagnosis capability can be regularly improved because the knowledge database can be regularly upgraded.

The first contribution of this paper is the description of the overall diagnosis process designed for the HIDS of [1]. The second contribution is the description of an implementation of this process, and in particular the representation of an alert within the knowledge database, its construction on-ground and its exploitation on-board. The third contribution is the description of an on-board implementation of the complete detection approach of [2], including anomaly detection and on-board diagnosis activities, as well as the performance evaluations performed on this realistic implementation.

Section II summarizes the overall HIDS approach and our main contributions. Section III discusses related work. The diagnosis process is presented in Section IV, with a specific focus on our implementation choices for two important steps of this process in Sections V and VI. An experimental setup of this approach is described in Section VII, while Section VIII describes experiments carried out on a real avionics computer integrating the complete HIDS (including detection and diagnosis), with associated performance evaluations. Section IX concludes and discusses future work.

## II. General HIDS Approach

The anomaly-based detection approach proposed in [2] is structured into two phases corresponding respectively to aircraft systems integration and operation phases. Figure 1 shows 1) the activities carried out during the integration phase, aimed at building a model of the legitimate behavior of the application; and 2) the activities performed during the operation phase to detect and characterize anomalies (i.e., behaviors that deviate significantly from the model built in the integration phase). The activities carried out on the ground are highlighted in green and those on-board in blue.

The *Static Security Analysis* aims to detect a corrupted or malicious binary received from an application supplier to be integrated. This analysis can be performed using existing anti-malware techniques to check the conformity between the binary and its documentation. The *SDA Modeling* and *SDA Validation* blocks are intended to build a model of the legitimate behavior of the application, referred to as "Security Domain of the Application" or SDA. The **Anomaly Detection** block performs real-time anomaly detection on-board the aircraft, identifying deviations from the SDA, and sends

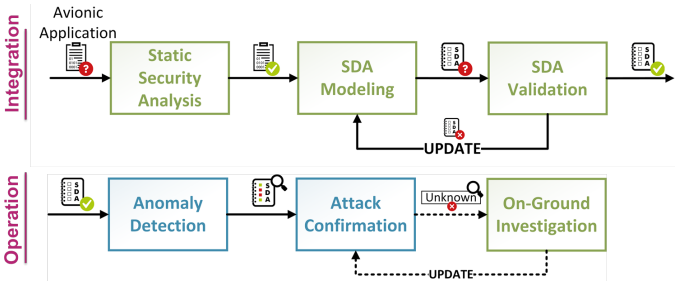



Fig. 1: Integration and Operation Phases

the anomalies to the **Attack Confirmation** block for further analysis. If this block is unable to indicate whether or not the anomalies correspond to an attack, they are sent to the ground for further investigation (**On-Ground Investigation** block).

This paper focuses on the design and implementation of the **Attack Confirmation** and **On-Ground Investigation** blocks. An alert is raised and the attack confirmation block is activated only when the number of anomalies detected during a given period exceeds a predefined threshold. Indeed, the role of the **Attack Confirmation** is to characterize these anomalies and check whether the characteristics correspond to a known event or not, recorded into a knowledge database. Note that this knowledge database records both attack patterns (just like in an antivirus database) and potential rare events that may be legitimate (like safety failures or false alerts).

The *Attack Confirmation* first computes these characteristics to associate a signature to a group of anomalies, which is then compared with signature patterns already recorded in the onboard knowledge database. As a result, the overall HIDS approach we propose is a mix of an anomaly-based detection system (**Anomaly Detection** activity) and a signature-based diagnosis system (**Attack Confirmation** activity).

The work described in [1] mainly focused on the definition of the SDA used for the anomaly-based detection system based on system calls analysis. This anomaly-based system was specifically designed to generate very few false alarms. Thus, the main objective of the **Attack Confirmation** is to provide on-board diagnosis about the alerts raised by the **Anomaly Detection** component of the HIDS. This diagnosis is supported by the **On-Ground Investigation** when the symptoms recorded do not match the information recorded in the database. As some very rare legitimate behaviors may have been missed during the definition of the SDA during the integration phase, the second objective of the diagnosis is then to identify these rare behaviors as benign or legitimate.

## III. Related Work

As stated in Section II, the detection system explored in our work is based on system calls analysis. Seminal work on anomaly-based intrusion detection analysis based on system calls has been published by [3] ; more recent works include e.g., [4] and [5]. In the following, we focus on misuse-based intrusion diagnosis, as it is the scope of this paper. We first

present related work on feature extraction and then on machine learning techniques for attack classification.

Feature extraction for host-based intrusion detection system using system calls has been mainly studied in the context of malware detection. The method proposed in [5] has been adapted to signature-based IDS, e.g., in [6] and [7] which use the most frequent $n$-gram system calls patterns to build a signature pattern of an attack. Some techniques based on the frequency of system calls have also been explored [8]. [9] extracts features with principal component analysis (PCA).

Various machine learning methods have also been used to perform attack classification. A recent survey [10] lists multiple classification models, such as $k$-nearest neighbours ($k$-nn), decision tree, support-vector machine (SVM) and artificial neural network (ANN) classifiers. Clustering techniques used on system calls have also been explored, e.g. [11].

However, not all these methods are compatible with the requirements presented in Section I. As per **O5**, the detection must be explainable: this excludes for the moment the use of some models, notably SVM, ANN and, among other ensemble learning techniques, random forests [12]. Note that, due to the recent interest of the scientific community in this matter, the interpretability of these models could be enhanced in the future. On the other hand, decision trees are straightforward to interpret, $k$-nn is explainable as it is based on a handful of similar observations and the log-likelihood computed by a naive Bayes classifier is a weighted sum of distances for each feature. The detection must also have a limited time and memory footprint (**O3** and **O4**); this disqualifies ANN that are generally computationally expensive. On the other hand, decision tree and naive Bayes classifier have a particularly reduced time and memory footprint. Furthermore, an important feature that is mandatory for these critical systems and is implicit in requirement **O5** is the possibility for the detection system to reject the classification and assign an "unknown" label to the data. This feature, called "reject option", is not embedded in all models. It can be easily added to probabilistic models such as the naive Bayes classifier. It can also be integrated in SVM [13], ANN [14] and $k$-nn [15]. Ad-hoc reject option can be added to decision trees [16] using Gaussian mixture model.

| Model | Explainable | Reject option | Time and memory footprint | Classification effectiveness |
|---|---|---|---|---|
| ANN | No | None | High | Excellent |
| SVM | No | Embedded | Medium | Very good |
| Random forest | No | None | Medium | Very good |
| Decision tree | Yes | Ad-hoc | Low | Good |
| Naive Bayes | Yes | Embedded | Low | Good |
| $k$-nn | Yes | Embedded | Medium | Good |

TABLE I: How well the models match with the requirements

A summary of the requirements fulfilled by each model is presented in Table I. The most suitable models in our context are the decision tree, the naive Bayes classifier and $k$-nn. In the following we make the choice to use the naive Bayes classifier as it fulfills all the requirements (explainable classification, reject possibility, very reduced time and memory footprint) and generally yields good classification results.

## IV. Diagnosis Approach

The overall diagnosis approach proposed in this paper is described by Figure 2. The **Attack Confirmation** is intended to provide an on-board diagnosis (deciding whether an alert corresponds to a real attack, a safety-related event or a false alert), while the **On-Ground Investigation** aims to analyze unrecognized alerts on the ground, in order to update further the knowledge database used for **Attack Confirmation**.

The **Attack Confirmation** activity is based on three components. First, **Signature Extraction** consists in computing some characteristics about the alert raised by the **Anomaly Detection** activity, in order to create a signature. This signature is then compared with entries of an onboard knowledge database (**Knowledge Database Search**). This knowledge database contains a list of signature's patterns, each pattern being associated to a label. Each signature pattern is intended to represent all known signatures for a specific attack, a safety event or a false alert. The last component (**Alert Message Sending**) selects the information to be recorded depending on the label of the current alert and the raw alert data, and builds an alert message based on this information. The reaction to such alert messages is out of the scope of this paper and is represented by the **On-Board Security Alert Management** activity. In practice, this could consist e.g., in i) storing the message for further analyses or to correlate the alert with other information monitored by the system, ii) applying an automatic reaction, or iii) suggesting recovery procedures to the crew. Finally, the alerts that are not recognized in the knowledge database (labeled as *unknown*) are sent to the ground for further investigation (**On-Ground Investigation**).

### A. Signature extraction

Different types of data may be used in order to define the signature of an alert. In avionics, the characteristics defined in the signature must be sufficient to differentiate three cases: 1) the alert corresponds to a real attack, 2) the alert corresponds to a safety-related event, and 3) the alert corresponds to a false alert. A real attack should be characterized by a real impact on the application, for example on its execution flow, its configuration, its permissions, or its registers usage. A safety-related event also causes a specific impact on the application's behavior. However, this impact should correspond to a known event, be handled by a safety monitor and raise a safety alert. Finally, no significant impact should be observed for a false alert and the application state should be normal. As a consequence, these three cases can be differentiated by means of: 1) the correlation of the security alert with a safety alert; this correlation process is deported directly to the **On-board Security Alert Management** activity, as this correlation might be done at the avionics suite level; 2) the state of the application's environment; it may be represented through contextual data or some execution data like the return address stored in the stack, to detect a misuse of the application's code for

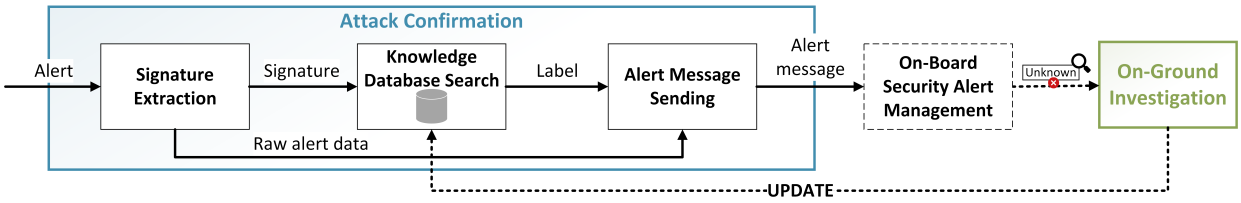



Fig. 2: Diagnosis approach

example, and 3) the characterization of the deviations between the observed and the normal behavior; it may be assessed through the distance between the anomalies raised and the model, the variability of such anomalies, or their occurrence time. More detailed information and concrete examples of these characteristics are provided in Section V, focusing on the implementation of this important step of our approach.

### B. Knowledge database search

Once the signature of the alert is computed, it is compared to the entries in the knowledge database (constructed during the on-ground investigation, see Section IV-D). These entries might correspond to real attacks (just like in an anti-virus database), a safety event, or a false alert. Each entry is composed of a label and a signature pattern. For each entry, the signature pattern aims at representing all known signatures for a specific attack or safety event or false alert. Each signature provided by the signature extraction step is compared to all the signature patterns of the knowledge database (by means of a distance metric) in order to identify if the signature is sufficiently close to one of these patterns (expressed by a log-likelihood score). If no entry matches the signature then the signature is assigned the label *unknown*. If multiple entries correspond to the signature, the most likely label is associated to the signature, based on the log-likelihood score. Information on this log-likelihood score and the algorithm used to efficiently search the database is provided in Section VI.

### C. Alert message elaboration and sending

Once a label has been associated to the signature of the alert, the last step of the **Attack Confirmation** consists in building the alert message and sending it to the **On-Board Security Alert Management** entity. This message can be constructed differently depending on the label associated to the alert. If the label is known, the message can be composed of the label and the signature only, in order to minimize the memory usage. On the contrary, if the label is *unknown*, it is important to provide more information to help the **On-Ground Investigation**. For example, the message could be composed of the raw data composing the alert (so-called *raw alert data* in Figure 2) and additional contextual data.

In practice, the message could be composed of the label only, and additional data could be stored directly in non-volatile memory by the HIDS (for example, the raw alert data, the signature, or the log-likelihood score).

### D. On-Ground investigation

The activities performed on the ground are represented by Figure 3. The alerts received from an aircraft fleet, and associated data, are aggregated on the ground for further investigation in order to update the knowledge database. Unknown alerts are investigated by an expert using associated alert data and the binary of the application incriminated, in order to attach a label to it. In case of an attack, the malicious code is stored in a dedicated database. The signature and associated label are then stored in an alert signatures database. This signature database can also be populated using 1) known alerts raised by the aircraft fleet, and 2) alerts generated on ground using the malicious code database.

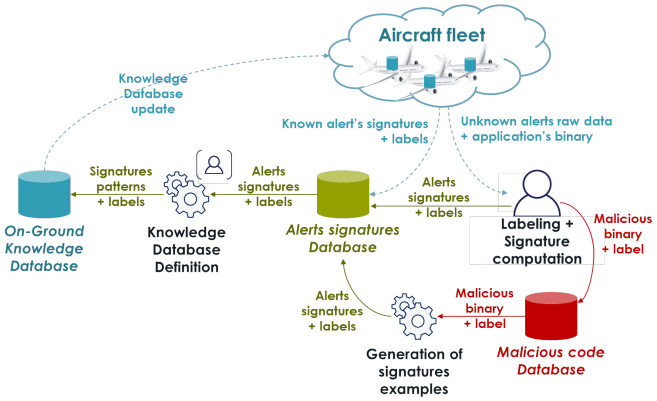



Fig. 3: Knowledge database construction

The alert signatures are generalized into a signature pattern, in order to construct the on-ground knowledge database. The alert signatures database contains many examples of signature for a same label. On the contrary, a signature pattern in the on-ground knowledge database identifies a set of signatures with similar characteristics, to which a specific label is associated. Finally, the on-ground knowledge database is used to update the knowledge database embedded inside each aircraft.

The following sections V and VI describe the implementation choices we made for the two main steps of our diagnosis approach investigated in this paper: the signature extraction and the knowledge database search.

## V. Signature Extraction

Taking into account the requirements outlined in section IV.4 for signature extraction, we have selected 18 characteristics listed in Table II to implement the signature of an alert.

TABLE II: Characteristics used as alert's signature

| Name | Description |
|---|---|
| $dist_{total}$ | Sum of distances between anomalies and SDA model |
| $dist_{min}$ | Min distance between anomalies and SDA model |
| $dist_{max}$ | Max distance between anomalies and SDA model |
| $nb_{close}$ | # of anomalies close to SDA model (distance less than specific threshold, fixed to 20 clock ticks) |
| $nb_{unknown}$ | # of anomalies due to unknown sequence |
| $nb_{ra,i}$ | # of different return addresses (*ra*) of level $i$, $i \in \{1,2,3,4\}$ |
| $nb_{ra\_global}$ | # of unique return addresses |
| $nb_{consec}$ | Max # of consecutive anomalies |
| $nb_{slot}$ | Max # of anomalies in one execution slot |
| $nb_{groups}$ | # of different abnormal sequences |
| $nb_i$ | Occurrences of communication-related API call n°$i$, $i \in \{1,2,3,4,5\}$ |

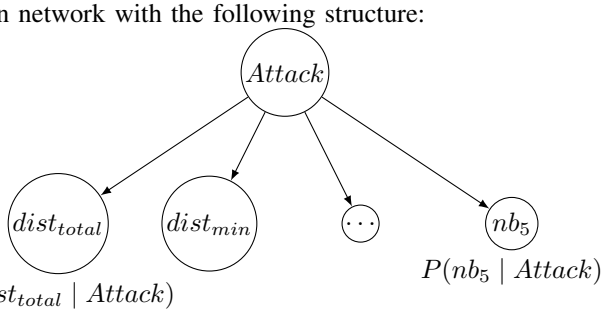
$dist_{total}$, $dist_{min}$, $dist_{max}$, $nb_{close}$, and $nb_{unknown}$ measure the distance between the anomalies and the SDA model. $nb_{close}$ represents the number of anomalies that are considered close to the SDA model, by means of a distance that is computed using a predefined threshold. $nb_{unknown}$ reports the number of anomalies with a distance of $-1$, representing abnormal sequences. Characteristics $nb_{ra,i}$, $nb_{ra\_global}$, $nb_{groups}$, $nb_i$ represent relationships between the anomalies, either regarding the location of the malicious code represented by the number of the different return addresses of level $i$ ($nb_{ra,i}$) and the number of unique return addresses ($nb_{ra\_global}$), or the variability of the anomalies represented by the number of different abnormal sequences ($nb_{groups}$) and the number of occurrences of API call $i$ ($nb_i$). Finally, the chronology of anomaly occurrences is captured through the maximum number of consecutive anomalies observed ($nb_{consec}$) and the maximum number of anomalies raised during the same execution slot ($nb_{slot}$). These characteristics are computed using the raw data attached to each alert. The relevance of these 18 characteristics is further investigated in Section VIII, in which some experiments are carried out.

## VI. Knowledge database search

The knowledge database construction consists in generalizing the signatures with a same label inside a same signature pattern. During the knowledge database search, a new signature is compared to all signature patterns of the database to identify whether it is sufficiently close to the patterns or not, based on its log-likelihood score. If all log-likelihood scores are below a threshold (that may be different for each pattern) then the *unknown* label is returned. Otherwise, the label corresponding to the signature pattern that maximizes the log-likelihood is returned.

In this implementation, we use the Naive Bayes classifier to perform the knowledge database searching, as explained in Section III. To compute the log-likelihood score between a signature and a pattern, the Naive Bayes classifier computes the probability for a signature to match the pattern. This probability is based on the mean and standard deviation of each feature observed from the learning data.

A naive Bayes classifier can be expressed as a simple Bayesian network with the following structure:




The node labelled "Attack" is the parent of the other nodes, each one corresponding to a characteristic, with he probability distribution in case of an attack.

As a consequence, a signature pattern $p$ is composed of the mean and standard deviation for each characteristic $c$ (respectively $\mu_{p,c}$ and $\sigma_{p,c}$) and a minimal log-likelihood score to reach ($min\_score_p$). A minimum value of $10^{-4}$ is also defined for the standard deviation. The mean and standard deviation for each feature are computed from a set of training signature examples of the same label. Then, the Naive Bayes log-likelihood is computed on each example, and the $min\_score_p$ is defined as the value of the 10th percentile observed from the distribution of these log-likelihoods.

This process is applied to each group of signatures with the same label, in order to obtain one signature pattern for each label inside the knowledge database. In practice, it could be interesting to provide multiple signature patterns for one label, for example if the overall behavior of anomalies is very different from an attack version to another. In this case, a first unsupervised classification may be performed on the signature examples with a same label to automatically define such multiple signature patterns. This has been implemented using a $k$-means clustering algorithm, with the number of clusters manually specified for each label.

Finally, the knowledge database search consists in computing the log-likelihood score between the new signature $x$ and each signature pattern $p$ in the knowledge database. Assuming that all features follow a normal distribution, the score (denoted $LL$) is computed as follows:

$$LL(p \mid x) = \log P(x \mid p) = A_p + \sum_c B_{p,c}(x_c - \mu_{p,c})^2$$

$c$ is a feature, $x_c$ its value for signature $x$, $\mu_{p,c}$ is the mean of the distribution of $c$ for the signature pattern $p$, $\sigma_{p,c}$ is the standard deviation of $c$ for $p$, $A_p = -\sum_c \frac{1}{2}\log(2\pi\sigma_{p,c}^2)$ and $B_{p,c} = -1/(2\sigma_{p,c}^2)$. $A_p$ and $B_{p,c}$ are constant and can be pre-computed to speed up the on-board processing. Since the log-likelihood is a weighted sum, it is easy to estimate the contribution of each feature that leads to the classification.

In order to take into account the possibility of an *unknown* label, we assign a default score $min\_score_p$ to the log-likelihood score, for each pattern. $min\_score_p$ can be interpreted as a confidence threshold: if no signature pattern matches the anomaly with at least its own log-likelihood $min\_score_p$, then we can't conclude on the signature label.

## VII. Experimental setup

The HIDS has been implemented with the diagnosis component as shown by Figure 4. This experimental setup is composed of three different hardware components: the target (a real avionic computer), the controller (a computer used to communicate with the target), and a prototyping station (a computer used to test the diagnosis capacity). This section describes this experimental setup, whose main objective is to evaluate the accuracy of the diagnosis on the prototyping station. The evaluation of the resource consumption overhead is evaluated based on the embedded implementation of the complete HIDS as described further in Section VIII-C.

### A. Target description

Two main partitions run on the target, namely App_HMI and HIDS. App_HMI is an aircraft application that displays information about the flight (like the speed and the altitude). The HIDS partition runs the **Anomaly Detection** and **Attack Confirmation** activities. The target uses only one core, and the execution of the partitions is scheduled periodically as shown in Figure 5. Every 50ms, App_HMI is executed during 6.7ms and the HIDS is executed during 3.6ms. On Figure 5, P2 represents other partitions that are also scheduled.

*1) SDA Monitor:* At runtime, the ARINC 653 API calls performed by App_HMI are monitored by the SDA Monitor. The SDA Monitor is implemented inside the RTOS by means of 53 assembly instructions (212 bytes). It intercepts the communication-related API calls performed by App_HMI and logs for each API call three types of information into a dedicated memory area (Log Frame): its ID, a timestamp, and four levels of return addresses (stored in the stack). The IDs and timestamps are computed in order to detect an abnormal behavior in the timed sequences of the calls performed by App_HMI (**Anomaly Detection**). The return addresses are only used for **Attack Confirmation**. Indeed, this information is useful to investigate the state of the application when an anomaly is detected, by analyzing the successive functions called before performing a particular API call.

*2) Anomaly Detection:* In this implementation, the SDA model used to detect anomalies is a Timed Automata[1] representing the authorized sequences of communication-related API calls and their expected duration. An anomaly is characterized by an unknown sequence or an abnormal duration of the sequence. In case of an abnormal duration, the distance between the closest expected duration and the observed duration is computed. In case of an abnormal sequence, this distance is set to $-1$. If too many anomalies are detected in a given time frame, an alert is raised. This threshold is defined by the SDA model and has been set to 10 for this implementation.

*3) Attack Confirmation:* This component associates a label to the current alert by means of the following process: 1) all the information of the Log Frame associated to the 10 anomalies of the alert are used to compute the alert's signature; 2) this signature is compared with the different signature patterns already defined in the embedded knowledge database, in order to identify the label for this current alert; and 3) the suited message alert associated to this label is sent.

For our implementation, some off-line experiments were first performed on the prototyping station in order to assess the efficiency of the diagnosis algorithm, by estimating the most relevant information to compose the signature and the most relevant algorithm to compare this signature to the signature patterns of the knowledge database. During these first experiments (described in Sections VIII-A, VIII-B and VIII-C), the prototype was only used to get realistic data to study. The complete implementation of the **Attack Confirmation** was performed in a second step in order to evaluate its resource consumption in real conditions (see Section VIII-D).

### B. Controller

The controller interacts with the target using a debugger. A GDB[2] server is instantiated on the target. It is directly attached to the RTOS, so that a breakpoint preserves the real-time execution of the target and its partitions.

*1) Attack Injection Tool:* This tool is used to perform mutations inside the monitored application in order to emulate a malicious modification of the application's binary before its loading inside the target. Because of the lack of real examples of attack in avionics, this tool, detailed in [17], has been developed in order to provide application's anomalous behavior examples that are representative of a potential attack. In this paper, the attack injection tool is used to generate attack examples by using specific parameters to represent attacks with a real impact on App_HMI. In practice, the modifications are performed using GDB at the initialization of the target. Four attacks have been implemented and are described in Table III. Attack *Modify_NAME* changes the value of a displayed field. The other attacks target specific functions F1, F2 and F3 that are regularly invoked inside the application, and provoke either a modification of a displayed field (*Skip_F1*) or a denial of service on the display (*Skip_F2* and *Skip_F3*). For each one, different versions have been defined by varying the duration and frequency of the malicious payload activation. For each attack, the attack was activated either at the beginning of the application for a short duration, or later with a longer duration. For *Skip_F1* and *Skip_F3* attacks, the attack was activated at each call of the function, or at one call over two. This frequency of activation was more explored with the *Skip_F2* attack, and vary from $1/1$ to $1/10$ of function calls skipped.

*2) Collector:* The collector is used firstly to collect the anomalies provided by the **Anomaly Detection** when an alert is raised. This raw data are stored inside a file on the controller, which is then used on the prototyping station. To collect this data, a breakpoint is set up inside the **Attack Confirmation** code. When the breakpoint is hit, a debug script logs a description of the current experiment (ID and version of the

[1] the results obtained in [1] showed that this model presented excellent performances in our context

[2] GNU Debugger: https://www.gnu.org/software/gdb/

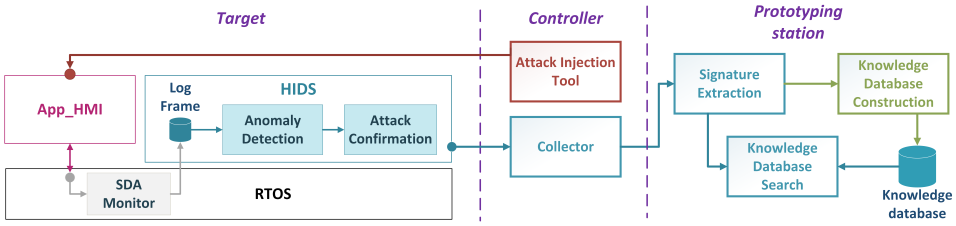



Fig. 4: HIDS prototype implemented, including diagnosis activity


Fig. 5: Real-time execution scheduling on the target

TABLE III: Attacks defined to test the diagnosis approach

| ID | Name | #versions | Description |
|---|---|---|---|
| 1 | Modify_NAME | 3 | Replace displayed field value by "PWN" |
| 2 | Skip_F1 | 5 | Suppress calls to F1 |
| 3 | Skip_F2 | 7 | Suppress calls to F2 |
| 4 | Skip_F3 | 4 | Suppress calls to F3 |

attack injected) and the raw data of the anomalies that led to the alert ([ID, timestamp, return address 1, return address 2, return address 3, return address 4] of each API call composing the abnormal timed sequence and the distance between the abnormal timed sequence and the SDA model, for each of the 10 abnormal timed sequences registered). An example of the raw data for each anomaly included in the alert is given below.

```
alert = {
# Anomaly n°i
ai = {log1 = {ID = 5, timestamp = 416798378,
        ra_1 = 0x10000000, ra_2 = 0x10001000,
        ra_3 = 0x10002000, ra_4 = 0x10003000},
      log2 = {...},
      distance = 50}
```

Note that during one experiment, many alerts may be raised.

Secondly, the collector is used to evaluate the execution time of the overall HIDS partition. Two breakpoints are set up at the beginning and at the end of the HIDS partition and are used to log the current target timestamp. The difference between the two timestamps is used to evaluate the execution duration of the HIDS partition.

### C. Prototyping station

The prototyping station is used to evaluate the accuracy of the Attack Confirmation in order to select the best solution to implement on the target. Three components have been developed: the signature extraction, the knowledge database construction, and the knowledge database search (described in Sections V and VI). At the end of this prototyping step, the chosen solutions for the signature extraction and knowledge database search are implemented on the target, while the knowledge database construction is carried out on-ground.

## VIII. Experimentation

The prototype previously described has been used for multiple experiments. First, this section presents the dataset used for these experiments. Then, the signature format proposed in Section V is analyzed to validate select the most relevant characteristics used for signature extraction. The next experiment aims to evaluate the accuracy of the diagnosis approach on the dataset (based on the algorithm presented in Section VI, in the case of known and unknown alerts. Finally, this section presents a full embedded implementation of the HIDS and an evaluation of the associated resource consumption.

### A. Dataset

The dataset is composed of 1208 raw alert data examples sent from the **Anomaly Detection**. Five classes are represented through these examples, with the following distribution: 246 false alerts (artificially generated), 214 alerts "Modify_NAME" attack (ID n°1), 270 alerts "Skip_F1" attack (ID n°2), 219 alerts "Skip_F2" attack (ID n°3), and, 259 alerts "Skip_F3" attack (ID n°4). Due to the lack of safety mechanisms implemented on the target used in the prototype (target under development), our experiments did not include examples with safety-related alerts. As a preprocessing step, an alert's signature is computed for each raw alert data of the dataset. In the following, the term *dataset* refers to the set of 1208 signatures.

### B. Characteristics selection

The first experiment evaluates the relevance of the characteristics proposed to implement the signature of an alert. This is done by 1) analyzing the dataset characteristics and 2) exploring several classification algorithms to check their ability to separate the dataset into classes. The Weka tool [18] has been used for these first experiments.

A Principal Components Analysis showed that five characteristics have a very high correlation (between 0.89 and 0.98): $nb_{ra,2}$, $nb_{ra,3}$, $nb_{ra,4}$, $nb_{ra\_global}$, and $nb_{groups}$. This correlation does not depend on the type of class considered (real attack or false alert). Therefore, it may not be necessary to consider all these characteristics for the signature.

An unsupervised learning algorithm was also applied to the dataset to check its ability to separate the data correctly. The $k$-means algorithm was used to separate the dataset into 5 clusters (i.e. as many clusters as the number of different classes).

Using the 18 characteristics proposed for a signature, the algorithm presented difficulties in differentiating attacks *Skip_F1* and *Skip_F3*. This resulted in 82.53% of well-classified alert examples. By removing the four characteristics correlated with the $nb_{groups}$ (i.e. $nb_{ra,2}$, $nb_{ra,3}$, $nb_{ra,4}$, $nb_{ra\_global}$), the algorithm produced good results (95.36% of well-classified examples). Accordingly, the proposed characteristics should be sufficient to differentiate the 5 classes in the dataset.

A second classification algorithm was applied to the dataset to validate the previous results and analyze the most important characteristics of the signature. A decision tree (supervised learning) has been learned to differentiate classes within the dataset. Using a 10 fold cross-validation, the resulting decision tree was able to correctly classify 95.94% of the examples. Also, the resulting decision tree showed the importance of distance-related characteristics ($dist_{total}$, $dist_{min}$, $dist_{max}$, $nb_{close}$) and the API calls impacted ($nb_i, i \in \{1,2,3,4,5\}$). Indeed, the resulting decision tree is mainly based on these characteristics to apply its classification.

To conclude, these experiments allowed us to validate the relevance of the 18 characteristics proposed to implement the signature of an alert, and to select the most important ones. As a result, only 14 over the 18 characteristics are used in the additional experiments presented later in this paper ($nb_{ra,2}$, $nb_{ra,3}$, $nb_{ra,4}$, and $nb_{ra\_global}$ are removed). These characteristics are stored in a signature database with a corresponding label, extracted from the description of the raw data file (ID and version of the attack injected). A subset of the signatures are used to build the knowledge database, and the remaining signatures are then used to test its accuracy, as described in the next Section. Also, these first experiments show that it is difficult to obtain 100% of correctly-classified examples using the proposed signature definition. The use of additional contextual data in the signature could help to improve the results. In practice, since multiple alerts are raised during an attack, a lower accuracy may be sufficient to provide relevant on-board diagnosis.

### C. Diagnosis accuracy evaluation

The experiments presented in this section aim to evaluate the effectiveness of the proposed diagnosis, before implementing it inside the embedded HIDS. The knowledge database is automatically constructed using the dataset presented in Section VIII-A. The signature format consists of the 14 characteristics selected after the dataset analysis (Section VIII-B).

Two experiments were conducted to assess the accuracy of the knowledge database: 1) based on the number of examples used to construct the database, and 2) in the case of an unknown class.

Accuracy is assessed according to the process described in Listing 1. First, the dataset is split into a training dataset and a testing dataset with a given ratio (from 5/95% to 80/20%). To consider a class as unknown, the corresponding examples are removed from the training dataset (and not from the testing dataset). The training dataset is then used to build the knowledge database, while the testing dataset is used to test the accuracy of the knowledge database search. The final score is computed as the ratio of well-classified examples from the testing dataset. The definition of a well-classified example depends on the representation of the class in the training dataset. If the class is represented in the training dataset, a correctly-classified example for this class is an example that is classified with the right label. If the class is not represented in the training dataset, a correctly-classified example for this class is an example that is classified with the label *unknown*.

```
# Load 1208 signatures examples
dataset = load("signatures.csv")
nb_clusters = {"False Alert":1, "Modify_NAME":2,
↪  "Skip_F1":2, "Skip_F2":1, "Skip_F3":1}
results = list()
for seed in range(20):
    for ratio in range(0.05, 0.80, 0.01):
      # Split the dataset with a given ratio
      train, test = split(dataset, ratio, seed)
      # Optional: remove classes from training set
      train.remove(unknown_classes)
      # Knowledge database construction
      kdb = build_kdb(train,nb_clusters)
      # Knowledge database search
      score = search_kdb(test,kdb)
      results.append(score)
```

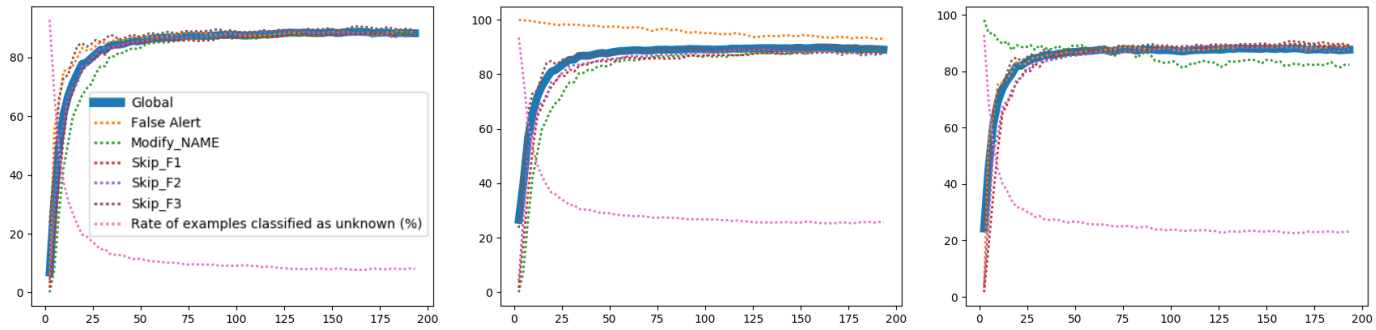
Listing 1: Accuracy evaluation process

The *nb_clusters* variable, in Listing 1, defines the number of signature patterns per class. In this experiment, the analysis of the examples from the classes *Modify_NAME* and *Skip_F1* showed two different behaviors depending on the version implemented (regarding respectively the moment of the attack's activation, and its frequency). This leads to the use of two signature patterns for each of these two attacks, represented through the value of the *nb_cluster* variable.

This experiment was run 20 times with different ratios between the training and testing dataset (represented by the loop over the *seed* variable). The results are presented in Figure 6, according to the classes represented in the training dataset. In sub-figure 6a, the 5 different classes are represented in the training dataset. For the remaining 5 sub-figures, one among the five classes is missing in the training dataset. For example, in sub-figure 6b, the False Alert class is considered to be correctly-classified if the label attached to it is *unknown*.

For the six experiments, the accuracy looks asymptotic regarding the number of signature examples used to build the knowledge database. When at least 70 examples of each class are used to build the knowledge database, the accuracy remains stable, with a similar value between 87% and 90% for each experiment. This number of examples can be reached quickly considering that multiple alerts are raised during an attack. Note that, if necessary, some signatures can also be artificially generated on the ground using the malicious code database, as mentioned in Section IV-D.

In figure 6a, the accuracy reaches 87% when at least 70 examples of each class are used to build the knowledge database. It may be explained by the introduction of the




(a) All classes represented in training dataset (b) No ”False Alert” in training dataset (c) No ”Modify_NAME” in training dataset

(d) No ”Skip_F1” in training dataset (e) No ”Skip_F2” in training dataset (f) No ”Skip_F3” in training dataset

Fig. 6: Results, depending on the classes represented in the training dataset (x-axis: Number of examples used for the knowledge database construction (per label); y-axis: Well-classified testing examples (%))

capacity to attach the *unknown* label to the examples, implemented through the $min\_score_p$ value inside the signature patterns. Indeed, around 10% of the examples were wrongly classified as *unknown*. Even though this capability induces a loss of accuracy when all classes are known, it significantly contributes to the good accuracy on the other experiments, when a class is not represented during training. In the end, only 3% of the signatures were assigned to a wrong pattern. Finally, on figure 6b and 6c, it seems that the accuracy on the class not represented in the training is decreasing with the number of examples used for the knowledge database construction. In fact, the increase in the number of examples for the training may introduce more permissive signature patterns. As a consequence, it is easier to mistakenly consider examples from an unknown class as an already known class. This problem could be solved, for example, by increasing the number of signature patterns for a given class.

### D. Resource consumption evaluation

This last section presents the experiments carried out to evaluate the resource consumption of the embedded HIDS, including the on-board diagnosis. First, the embedded implementation of the complete HIDS (detection and diagnosis) is presented as well as its final size in term of memory. Then, the timing overhead impact of the SDA Monitor on the existing ARINC 653 API calls is evaluated. Finally, the real time consumption of the HIDS partition is evaluated.

*1) Embedded implementation:* To evaluate the resources consumption of the overall HIDS, the HIDS partition has been updated to perform on-board diagnosis, as illustrated in Figure 7. The three components **Signature Extraction**, **Knowledge Database Search** and **Alert Message Sending** have been integrated inside the HIDS partition. **Signature Extraction** computes the signature relative to an alert, when too many anomalies are raised by **Anomaly Detection** in a given duration. This signature is composed of the 14 characteristics selected from the experiments presented in Section VIII-B. **Knowledge Database Search** component compares the extracted signature with the signature patterns recorded in its knowledge database, as implemented on the prototyping station. The embedded knowledge database is generated off-line with the complete dataset used for the accuracy evaluation experiments (e.g. 1208 alert examples distributed among 5 different classes). Also, two signature patterns have been set for each of the *Modify_NAME* and *Skip_F1* classes, leading to a knowledge database with seven entries. Finally, the label of the alert is used by the **Alert Message Sending**, which sends an alert message containing the label through the aircraft network.

The HIDS partition size is 49.4 KB (0.18% of the size of the monitored application App_HMI, 27.2 MB). This includes the anomaly detection part (log frame, SDA model and detection code) and the attack confirmation part (knowledge database and diagnosis code). The storage of additional information has not been implemented (it should be determined taking into account operational constraints), but it represents 524 bytes when an unknown alert is raised (storage of all raw alert

data and a timestamp) and 80 bytes when a known alert is raised (storage of the signature, the label and a timestamp), considering that each information is encoded on 4 bytes.

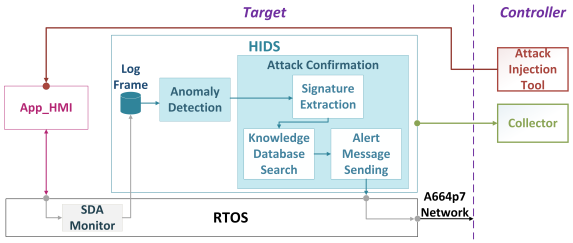



Fig. 7: Embedded implementation on the target

*2) SDA Monitor impact:* We measured the duration of the API calls performed by the App_HMI application, with and without the SDA monitor. More than 11000 captures of each experiment have been performed. An overhead of 6 clock ticks (+3.3%) is observed for the communication-related API calls (mainly impacted by the SDA Monitor), on the median value. It is very similar to the overhead measured in [1] (anomaly detection only, 5 clock ticks), even if more information is logged at each application API call. As expected, there is no impact on the other API calls.

*3) HIDS Partition duration:* The last experiment aims to measure the real execution time of the HIDS partition when an alert is raised. In this case, both **Anomaly Detection** and **Attack Confirmation** are executed and different attacks were injected. About 500 duration samples have been captured for this experiment. The median execution time of the HIDS partition is 0.059 ms. This represents 0.88% of the duration allocated to the monitored partition (App_HMI, 6.7 ms). Compared to the results described in [1] (0.014 ms for the HIDS partition with **Anomaly Detection** only), the introduction of the on-board diagnosis represents a high increase in terms of execution time, but is still very efficient in terms of the time allocated to App_HMI. These results are summarized in Table IV. Note that the worst case execution time has not been evaluated for the complete HIDS (detection and diagnosis). The execution time of the diagnosis part of the HIDS depends linearly on the number of entries in the knowledge database. It should remain very low given the very low probability of an attack on an aircraft. Also, the number of entries can be reduced using e.g., a decision tree [19].

TABLE IV: Resources consumption of the HIDS - Summary

| Partition | Code size | Execution time | |
|---|---|---|---|
| | | Total | API calls |
| App_HMI | 27.2 MB | 6.7 ms | - |
| HIDS_v1 (detection only) | 28.4 KB (0.11%) | 0.014 ms (0.21%) | +2.7% |
| HIDS_v2 (detection + diagnosis) | 49.4 KB (0.18%) | 0.059 ms (0.88%) | +3.3% |

## IX. Conclusion and Future Work

This paper presented a HIDS specifically designed for the avionics context, composed of an anomaly-based detection system and a signature-based diagnosis system. The anomaly-based detection system was presented and evaluated by [1], and was proven to be effective in detecting anomalous behavior of an avionic application. This paper focused on the signature-based system, designed to provide an on-board diagnosis of the alert raised by the anomaly-based detection system. This diagnosis is extremely important in the avionics context: an accurate diagnosis is the first step to provide an automatic reaction on-board in case of an attack.

The implementation of the diagnosis approach in the embedded HIDS and associated experiments has yielded good results in terms of the accuracy of the diagnosis and the associated overhead. In particular, between 87% and 90% of the alerts raised by the anomaly-based system are correctly diagnosed. The wrongly-classified alerts are composed of about 10% classified as *unknown* and 3% classified with a wrong attack label. In fact, the diagnosis proposed in this paper has been specifically designed to allow classification of *unknown* alerts. In the avionics context, a wrong classification could induce unexpected consequences. Therefore, it is far better to classify an alert as unknown than to mislabel it. The on-board implementation has shown low overhead in terms of resource consumption for the overall HIDS. The increase of resource consumption related to the integration of the diagnosis capacity still remains acceptable (the general objective being to use less than 5% of the available resources).

In practice, the main difficulties encountered while prototyping the avionics HIDS were related to the data sources (selecting relevant information to monitor without introducing complex instrumentation), the experimental environment (trade-off between application's availability, target's features, and target's novelty), and the multiplicity of competencies involved (modeling, configuration, integration, platform, application, tools).

Concerning the certification of our solution, even if there is no guidance available yet for Machine Learning-based solutions certification, we anticipated this need by selecting algorithms that should fulfill explainability and embeddability requirements. It should be noted that the criticality does not lie in the detection and diagnosis, but in the reaction that could be based on such diagnosis. The commercial deployment of the HIDS without automatic reaction should be easily achievable (no or low impact on aircraft safety), and should enable the evaluation of the solution in an operational environment before considering an on-board reaction.

To go further, we plan to test the overall HIDS approach on a more advanced environment (especially with the occurrence of safety-related events with realistic safety monitor, and with multiple and more complex applications exposing interactions between them). The diagnosis part should also be improved, for example by adapting the computation of the expected distribution of values, for each signature characteristic. Finally, we also plan to study the worst case behavior of our approach in terms of memory and execution time.

## REFERENCES


[1] A. Damien, M. Marcourt, V. Nicomette, E. Alata, and M. Kaaniche, "Implementation of a host-based intrusion detection system for avionic applications," in *2019 IEEE 24th Pacific Rim International Symposium on Dependable Computing (PRDC)*, Dec 2019, pp. 178–17 809.

[2] A. Damien, M. Fumey, E. Alata, M. Kaaniche, and V. Nicomette, "Anomaly based intrusion detection for an avionic embedded system," *Aerospace Systems and Technology Conference (ASTC), London, United Kingdom*, Nov. 2018.

[3] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Proceedings 1996 IEEE Symposium on Security and Privacy*. IEEE, 1996, pp. 120–128.

[4] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007, pp. 5–14.

[5] N. Hubballi, S. Biswas, and S. Nandi, "Sequencegram: n-gram modeling of system calls for program based anomaly detection," in *2011 Third International Conference on Communication Systems and Networks (COMSNETS 2011)*. IEEE, 2011, pp. 1–10.

[6] E. Aghaei and G. Serpen, "Ensemble classifier for misuse detection using n-gram feature vectors through operating system call traces," *International Journal of Hybrid Intelligent Systems*, vol. 14, no. 3, pp. 141–154, 2017.

[7] P. Deshpande, S. C. Sharma, S. K. Peddoju, and S. Junaid, "Hids: A host based intrusion detection system for cloud computing environment," *International Journal of System Assurance Engineering and Management*, vol. 9, no. 3, pp. 567–576, 2018.

[8] S. Das, Y. Liu, W. Zhang, and M. Chandramohan, "Semantics-based online malware detection: Towards efficient real-time protection against malware," *IEEE transactions on information forensics and security*, vol. 11, no. 2, pp. 289–302, 2015.

[9] G. Serpen and E. Aghaei, "Host-based misuse intrusion detection using pca feature extraction and knn classification algorithms," *Intelligent Data Analysis*, vol. 22, no. 5, pp. 1101–1114, 2018.

[10] M. Liu, Z. Xue, X. Xu, C. Zhong, and J. Chen, "Host-based intrusion detection system with system calls: Review and future trends," *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, pp. 1–36, 2018.

[11] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario, "Automated classification and analysis of internet malware," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2007, pp. 178–197.

[12] F. K. Došilović, M. Brčić, and N. Hlupić, "Explainable artificial intelligence: A survey," in *2018 41st International convention on information and communication technology, electronics and microelectronics (MIPRO)*. IEEE, 2018, pp. 0210–0215.

[13] G. Fumera and F. Roli, "Support vector machines with embedded reject option," in *International Workshop on Support Vector Machines*. Springer, 2002, pp. 68–82.

[14] Y. Geifman and R. El-Yaniv, "Selective classification for deep neural networks," in *Advances in neural information processing systems*, 2017, pp. 4878–4887.

[15] M. E. Hellman, "The nearest neighbor classification rule with a reject option," *IEEE Transactions on Systems Science and Cybernetics*, vol. 6, no. 3, pp. 179–185, 1970.

[16] P. X. Huang, B. J. Boom, and R. B. Fisher, "Hierarchical classification with reject option for live fish recognition," *Machine Vision and Applications*, vol. 26, no. 1, pp. 89–102, 2015.

[17] A. Damien, N. Feyt, V. Nicomette, E. Alata, and M. Kaaniche, "Attack injection into avionic systems through application code mutation," in *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)*, Sep. 2019.

[18] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[19] C. Kruegel and T. Toth, "Using decision trees to improve signature-based intrusion detection," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2003, pp. 173–191.