Implementation of a Host-Based Intrusion Detection System for Avionic Applications

Aliénor Damien, Michael Marcourt, Vincent Nicomette, Eric Alata, Mohamed Kaâniche

▶ **To cite this version:**

Aliénor Damien, Michael Marcourt, Vincent Nicomette, Eric Alata, Mohamed Kaâniche. Implementation of a Host-Based Intrusion Detection System for Avionic Applications. 2019 IEEE 24th Pacific Rim International Symposium on Dependable Computing (PRDC), Dec 2019, Kyoto, Japan. pp.178-17809, 10.1109/PRDC47002.2019.00048 . hal-03094199

HAL Id: hal-03094199
https://laas.hal.science/hal-03094199

Submitted on 4 Jan 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Implementation of a Host-based Intrusion Detection System for Avionic Applications

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

***Abstract*—Today, aircraft are protected by strong safety properties, qualified operators and process-based security measures. However, considering the recent evolution of in-flight services towards more connectivity, resource sharing and advanced entertainment functionalities, together with the increase of threats targeting embedded systems, the potential malicious modification of an aircraft application must be seriously considered for future systems. In this context, several solutions can be developed to improve aircraft security. In particular, Host-based Intrusion Detection Systems (HIDS) are relevant to deal with targeted threats such as an insider attack. This paper presents the specific constraints of building an HIDS onboard an aircraft, and discusses some relevant solutions that satisfy these constraints. These solutions are evaluated in terms of detection efficiency and resource consumption in order to select the solution that allows the best trade-off between efficiency and performances. The implementation of this solution on an embedded avionic computer is also described.**

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

I. Introduction

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 leads to a larger attack surface. With the continued increase of threats targeting embedded systems, the potential malicious modification 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 strict requirements inherent to embedded avionic computer systems and applications, in particular in the context of Integrated Modular Avionics (IMA) based architectures. An IMA system is organized as a network of computing modules, each supporting several applications, possibly of mixed criticality levels. The execution of mixed-criticality software on the same module is supported by space and time segregation mechanisms in compliance with the ARINC 653 standard. Each application is composed of one or several partitions. Each partition is allocated statically and periodically an execution slot, as well as memory resources protected by the underlying operating system.

The following requirements and constraints must be considered in the design of an HIDS aimed at detecting potential attacks during the execution of avionic applications:

- Aircraft systems may use proprietary Real-Time Operating Systems (RTOS) with custom file systems, static configurations, and non-standard instrumentation toolkits. In this study, we consider the RTOS as trusted.
- The detection of a security incident implies a reaction from the crew, the operators on the ground, or directly an automatic reaction of the embedded computer in future systems. Because such reaction must not have an impact on the flight safety, there are strong requirements on the detection accuracy and the alerts raised should be deterministic and explainable. Particularly, we consider that no false alerts should be raised in order to build a strong confidence in the alerts raised.
- Software development is expensive, especially for critical functions e.g. DAL A software such as RTOS, and for software updates (deployment cost). Generally, a daily update cannot be considered. Currently, the most frequent updates on an aircraft are every 28 days.
- As with any embedded system, the resources available are limited.
- Finally, avionic systems are real-time systems with periodically and statically scheduled applications. The HIDS should not disrupt the real-time execution of the monitored partitions. Moreover, some real-time properties must be guaranteed, such as the Worst Case Execution Time (WCET).

Many actors are involved in the development of an aircraft. In this study, we adopt the viewpoint of the Module integrators, i.e., the companies that perform the technical integration of the applications on avionic computers. It is assumed that they are responsible of the configuration of the HIDS and may not have access to detailed specifications of the applications they have to integrate (in the worst case, only the binary and the resources allocated to the application are available).

The design of the HIDS to be embedded onboard the aircraft should allow for an optimal tradeoff between the following requirements:

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
O6 Being efficient even on “black-box” applications

This paper proposes an anomaly-based HIDS adapted to the

IMA context that is aimed at fullfing the above objectives. Some preliminary concepts behind the proposed approach have been discussed in [1]. The main idea is first to build, during the integration phase, a model of the legitimate behavior of an avionic application, based on data collected by monitoring specific features during its execution (related to the application itself or its environment), and second, in the operational phase, to raise alerts when the behavior of the application exhibits significant deviations from this model. In [1], we investigated the use of OneClass Support Vector Machine learning model (OCSVM) to describe the normal behavior of applications using the observed sequences of API calls as inputs. The few experiments carried out with this model have shown very encouraging results in detecting anomalies, but the required resources for the model to be embedded on an IMA platform have not been evaluated.

In this paper, we revisit and significantly extend this work by: 1) exploring several alternatives to model the normal behavior of an application and 2) running a set of experiments in which these alternatives are tested and compared with different sets of data to be monitored. These alternatives are evaluated regarding their detection accuracy and their resource consumption, in order to select the most appropriate ones to be embedded in an aircraft. This paper finally details the implementation of a prototype of this HIDS on an avionic platform, as well as a performance evaluation of this prototype.

Section II discusses related work. An overview of the proposed approach is presented in Section III as well as different strategies for building the model describing the legitimate behavior of the monitored application. The evaluations of these alternatives in terms of detection accuracy and resource consumption are respectively presented in Section IV and Section V, while Section VI describes the embedded implementation details of the solution we consider the most appropriate to be embedded and evaluate its resource consumption in realistic conditions. Section VII concludes and discusses future work.

II. Related Work

This section first presents research work dealing with aircraft security measures, and more precisely related work about embedded IDS (Section II-A). Then, the main components of an HIDS, i.e., the data to monitor and the algorithm used to perform the detection based on these data, are also discussed in Sections II-B and II-C.

A. Security in Avionic Systems & Embedded IDS

Considering some recent attacks on embedded systems [2], [3], airworthiness regulations have evolved [4], [5], [6]. Today, avionics players have to consider on-board and ground infrastructure security. This evolution has been taken into account in the design of new aircraft. Perimetric defenses are implemented to divide the network into different domains [7], [8]. The systematic analysis of vulnerabilities when developing new platforms is also considered, as presented for instance in [9]. Aircraft also implement strong safety mechanisms that are historically designed to provide protection against accidental threats, and recently some security solutions have been proposed to cope with malicious attacks [10]. However, to the best of our knowledge, Intrusion Detection Systems (IDS) have not been implemented yet for aircraft systems. Such mechanisms would be useful to detect potential attacks exploiting unknown vulnerabilities and to provide additional protection mechanisms in the case of attacks not covered by the existing safety and other protection mechanisms.

A few studies have been published about IDS in embedded systems. For instance, [11] highlights some related constraints and challenges. [12] propose an IDS for embedded automotive architectures. The use and implementation of IDS on multi-core architectures for real-time embedded systems is investigated in [13]. Some studies proposed hybrid IDS to take advantage of both signature-based and anomaly-based techniques [14], [15]. Concerning avionics domain, [16] proposes an IDS aimed at monitoring avionic networks.

Our research focuses on a different approach, aiming at integrating a Host-based IDS (HIDS) into each avionic computer that is designed to monitor the behaviour of the hosted applications, using in particular avionic RTOS as a source of data. The design and implementation of such HIDS requires a preliminary selection of the data to monitor and of the algorithm used to perform the detection.

B. Monitored data for HIDS

Avionic data monitoring systems are generally focused on faults and failures. In IMA systems, this is managed by the Health Monitoring (HM) [17]. Three levels of HM are generally defined: process HM (managed by the application), partition HM, and module HM (managed by the RTOS). Faults include for example ARINC 653 error codes, missed deadlines, numeric errors, or illegal requests. Only a subset of the faults generates a message that is logged for further investigation by maintenance operators. These messages could be used as input data for data monitoring but the information provided is limited to messages that are already formatted. In addition, if the application is corrupted, the error messages generated by process faults could be altered.

Some platforms propose specific instrumentation tools that can be used by the module integrator to check the correct allocation of resources. The monitored information may include data on memory resource usage (data RAM), process stacks, main stack, error handler stack, non-volatile memory, or communication services memory. This system is non-intrusive and could be an interesting source for data monitoring, but the monitored data and associated tools are not standard (they depend on the execution platform), and the data are limited to resource allocation.

Other sources of data can be considered. In the studies related to embedded HIDS, the data sources used are memory usage [18], system calls distribution [19], execution time [13], or a subset of system calls [11]. In the survey published in [20], HIDS on IT systems are classified according to the source of the used data: System logs and audit data, windows registry data, file system monitoring, and process

and stored binaries. Notably, a section is dedicated to the system calls, as this data has been widely used to successfully detect anomalies. Syscalls can be observed in two main ways, through sequential features or frequency-based features. The survey also highlights some studies using additional data such as system call arguments or memory pointers.

In this paper, we use the ARINC 653 API calls [21] along with the execution time as input data. Indeed, system calls have been shown to be very efficient to detect behavior deviations and can be monitored at the OS level. Also, [13] shows that it is very relevant to monitor the execution time in a real-time system because of the periodic behavior of the observed applications. In our context, the ARINC 653 API calls timestamps can be directly monitored by the OS, in order to monitor the time execution of the application.

C. Anomaly Detection Techniques

Two types of intrusion detection techniques are generally distinguished: signature-based or anomaly-based. Signature-based HIDS have strong limitations to be embedded in an avionics context, such as frequent updates and their inefficiency to detect new or sophisticated attacks. Anomaly-based HIDS seem more appropriate, especially since we expect the false alarm rate to be reduced due to the static characteristics of avionic environment.

[22] proposes a classification of anomaly detection techniques at different levels, with a particular focus on Neural Network models. These models have proven to be efficient in detecting anomalies. However, they require high computing resources and are quite difficult to interpret. These limitations are hardly consistent with objectives O3 and O5 (real-time detection, and high confident and explainable results).

Other classification techniques have also been widely used to develop IDS [23], [24]. For example, [16], [25], [26] consider using OneClass Support Vector Machine (OCSVM) to detect network traffic anomalies, respectively on aircraft system, SCADA systems, and USB communications. This technique gives results that seem difficult to interpret (objective O5) but are very effective for detection, which is compliant with objective O3.

Also, [20] pointed out that Hidden Markov Models (HMM) have been used successfully to model system calls usage. These models are easier to explain but, since they are probabilistic, rare events such as safety-related events are not easy to model as part of the normal behavior model. A simplest model called timed automata was used in [27] to detect anomalies in the behavior of a Digital Video Broadcasting System, and in [28] to detect ATM frauds. Due to the deterministic behavior of such systems, the timed automata gave very good results. This simple technique is easy to explain and to interpret and can be very interesting in an avionics context. Also, it seems suitable to easily model and verify the timing behavior of ARINC 653 API call sequences.

Table I summarizes the characteristics of the anomaly detection techniques considered in this section, with respect to the objectives of an avionic HIDS. Objectives O1 and O2 are not represented, as they mainly depend on the implementation of the HIDS. Finally, the "OCSVM" and "Timed Automata" approaches are considered the best solutions in our context, for their promising results and real-time detection capability.

Table I: Anomaly Detection Techniques and HIDS Objectives

Anomaly Detection	O3	O4	O5	O6
Neural Network				X
OCSVM	X	X		X
HMM	X	X		
Timed Automata	X	X	X	X

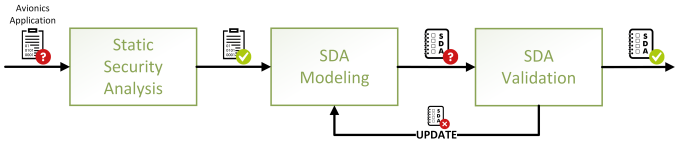

Figure 1: Integration Phase

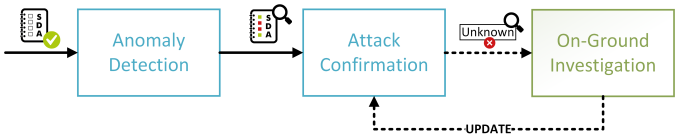

Figure 2: Operation Phase

III. Approach

The anomaly-based detection approach we propose, is structured into two phases corresponding to the integration and operation phases of aircraft systems. Figure 1 presents the activities carried out during the integration phase, aimed at building a model of the legitimate behavior of the application, and Figure 2 shows the activities during the operation phase, aimed at detecting anomalies (i.e., behaviors that deviate significantly from the model obtained in the integration phase). The activities carried out on the ground are plotted in green and those onboard in blue.

For each monitored application, the "Static Security Analysis" aims at detecting a corrupted or malicious binary received from an application supplier to be integrated. Two ideas are pursued to perform this analysis: 1) Use existing anti-malware techniques on the binary and 2) Check compliance 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, that is referred to as "Security Domain of the Application" or SDA. The "Anomaly Detection" block performs real-time anomaly detection onboard the aircraft, identifying deviations from the SDA, and sends the anomalies to the "Attack Confirmation" block for further analysis to reduce false alarms. If the "Attack Confirmation" block is not able to indicate whether or not the anomaly corresponds to an attack, it is sent to the ground for further investigation. This is represented by the "On-Ground Investigation" block.

This section focuses on "SDA Modeling" and "SDA Validation", proposes different solutions to obtain the SDA and a process to evaluate the detection accuracy of each solution. These steps are based on the Machine Learning process, composed of a preprocessing, training, and testing phases.

A. HIDS alternatives

The preliminary anomaly detection model presented in [1] is based on an OCSVM classifier using the ARINC 653 API calls performed by the applications as classifier inputs. More precisely, observed ARINC 653 API calls are classified according to their sequences and associated duration. The model provided interesting detection results, however two main limitations can be raised with respect to the possibility of running the model onboard: the amount of data logged at each execution slot of the monitored application may be very large, and, therefore, too many resources may be needed to process this data in real-time. In this section, we study different solutions to address these issues, combining different data monitoring strategies and different models.

Four data monitoring strategies are investigated. The **All API calls** strategy (used in [1]) consists in logging each API call with its corresponding timestamp. The **Communications only** strategy is similar, except that only services related to communications are logged. This means less impact on the monitoring, less data at each execution slot, faster detection, and easier implementation on legacy aircraft. However, there is also a loss of information that may decrease the detection accuracy. The volume of data processed should be lower compared to the **All API calls** strategy, or the same in the worst case. The **API calls frequency** strategy consists in computing the number of API calls performed during each execution slot of the monitored application. This monitoring is easy to implement, fast, and requires limited memory space, making it a good candidate to be embedded in an aircraft. Nevertheless, ignoring the timestamps of the API calls may significantly decrease the detection accuracy. Finally, the **API sequences frequency** strategy consists in logging, at each execution slot, for each possible sequence, the number of its occurrences along with its minimum, maximum and mean duration. The main advantage of this strategy is that it keeps information on execution time and requires limited memory. However, the additional computation cost per API call could be significant.

Three different models are considered here. **OCSVM** has been used in [1] to classify sequences but it should be more adapted to frequency-based data ("API calls frequency" and "API sequences frequency"). As a Machine Learning technique, it is very efficient to generalize data of a learning set. It is simple to implement and the detection is fast. The **Automata** model represents the API calls as states and the allowed transitions as edges. Such a model is particularly suitable for representing API call sequences. The **Timed Automata** model is an extension of the Automata model introducing allowed time intervals between two states. This combines the efficiency and explainability of automata with the representation of API call sequences duration.

Table II: Proposed HIDS Solutions

Solution	Monitor	Model
OCSVM_all	All API calls	OCSVM
A_all	All API calls	Automata
TA_all	All API calls	Timed Automata
OCSVM_comms	Communications only	OCSVM
TA_comms	Communications only	Timed Automata
OCSVM_API_freq	API calls frequency	OCSVM
OCSVM_seq_freq	API sequences frequency	OCSVM

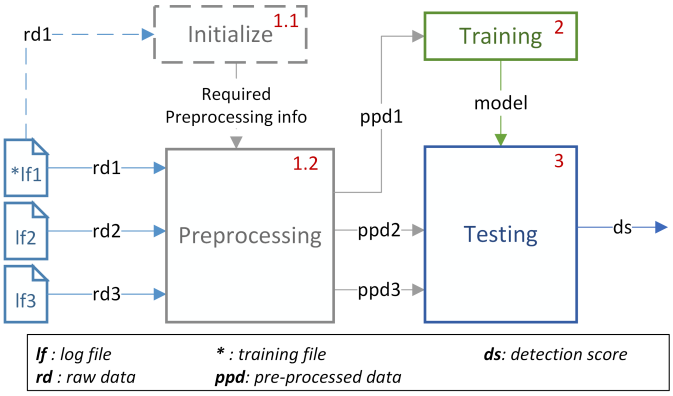

Figure 3: Process to Evaluate an HIDS Solution

Based on these different strategies for data monitoring and anomaly detection models, we propose 7 different HIDS solutions, as summarized in Table II. The OCSVM is evaluated on each data monitoring strategy proposed. The Automata and Timed Automata are only evaluated according to "All API calls" and "Communications only" strategies, as they are not adapted to the two other data monitoring strategies. The Automata model is not evaluated on the "Communications only" strategy, because we consider that the loss of information is too important compared to the solutions considering both the API calls and the duration of the data observed.

B. HIDS evaluation process

The process used to evaluate each HIDS solution is described by Figure 3 and Algorithm 1. It is composed of three phases. The evaluation process inputs consist of a set of log files containing the raw data observed during the execution of the application under study on the avionic platform, each file corresponding to one execution trace. The format of the raw data is a list of [API call ID, timestamp] logs. The log files are labeled as normal or containing an attack.

One normal log file is used as a training file to build the SDA model of the application (lf_1 in Figure 3). During the Preprocessing phase, the raw data are transformed into a set of features that are defined according to the type of models to be trained and tested. As detailed later, these features are different from one solution to another. The detection performance of the model resulting from the training phase is assessed during the testing phase which consists in evaluating each remaining

Algorithm 1: HIDS Evaluation Process

```
Input: log_files, labels
Output: score
Algorithm HIDS_Evaluation(log_files, labels)
1   training_lf, testing_lfs ← select_training_file(log_files)
    // Preprocessing
2   req_preproc_info ← initialize(training_lf)
3   for lf in log_files do
4       ppd[lf] ← preprocessing(lf)
    // Training
5   model ← train(ppd[training_lf])
    // Testing
6   for lf in testing_lfs do
7       decision[lf] ← predict(model,lf)
    // Detection score
8   return compute_score(decision,labels)
```

preprocessed log file not used during the training phase, as normal or containing an attack.

The *detection score* is computed by comparing the decision made by the HIDS and the label of each file. This score is defined as the Recall if the Precision is 100%, 0 elsewhere. The Precision and Recall are defined as follows (TP for True Positive, FP for False Positive and FN for False Negative):

$$Precision = TP/(TP + FP) \quad (1)$$

$$Recall = TP/(TP + FN) \quad (2)$$

This score represents the rate of detected attacks while no false alarms are raised. The objective to optimize precision is very important in our specific context, because a false alarm could have a significant impact on the pilot or on the flight safety. In order to provide an automatic reaction in the future, a high confidence in the alarm raised is needed (Objective O5).

C. Preprocessing Phase

This section presents the details of the preprocessing tasks performed on the raw data log files for the seven HIDS solutions previously defined. The format of each log is [API call ID, timestamp].

1) OCSVM_all, OCSVM_comms: The first preprocessing task is performed by the *initialize* function (see Algorithm 1) which consists in retrieving every possible API call sequence of length *n* from the training raw data file and assigning an index to each of them, using a sliding window of length *n*. Then, an output data vector is produced composed of 1) a one-hot encoding[1] of the sequence index and 2) the duration of the sequence. The duration is computed as the difference between the timestamps of the first and last log of the window. The output data vectors are then scaled. The size of the vector corresponds to the number of observed sequences. The same preprocessing is applied in the case of the "OCSVM_comms" solution considering only communication-related API calls.

2) A_all: The *preprocessing* function parses the input log file using a sliding window of length *n*, *n* being the length of the sequence considered. The output of the preprocessing of each window consists of the ordered API Call IDs of the observed sequence.

3) TA_all, TA_comms: Besides the preprocessing performed for the "A_all" solution, the duration of each observed sequence is also reported. In the case of the "TA_comms" solution, the same preprocessing is applied considering only communication-related API calls.

4) OCSVM_API_freq: The *initialize* function consists in retrieving every possible API call ID from the training raw data file and assigning an index to each of them. Then, the *preprocessing* function executes two phases. In the first phase, each log file is split into timing windows corresponding to one execution slot. In the second phase, one output data vector is generated per execution slot. The output data vector is composed of *m* counters (*m* being the number of API calls found in the training file), each one representing the number of times an API call is observed. When all logs are processed, the final output data vector is scaled.

5) OCSVM_seq_freq: The same *initialize* function than "OCSVM_all" is used. The *preprocessing* function executes two phases. The first phase is the same as in the "OCSVM_API_freq" solution. In the second phase, one output data vector is generated per execution slot. The output vector consists of $4 * m$ values, *m* being the number of sequences found in the training file. For each observed API call sequence index, the number of times it is observed during the execution slot is reported together with its minimum, maximum, and average duration. When each log has been processed, the final output data vector is scaled.

D. Training Phase

The implementation of the training phase depends only on the model used. In each case, the training is performed on a single normal preprocessed data file. Indeed, one file contains many examples of the periodic behavior of an application and should be sufficient to model it. Also, we want to limit the additional work of the Module Integrator to obtain raw data.

1) OCSVM: The OCSVM model defines an hypersphere characterized by a center *c* and a radius *R>0*, with a predefined margin to reduce false positives. The algorithm used to define this hypersphere is detailed in [29]. Four kernels are usually used with the OCSVM algorithm: linear, polynomial, radial basis function (RBF) and sigmoid. In this study, the RBF kernel has been used.

2) Automata: The model is built as a list of possible API call sequences. This list corresponds to the set of transitions in the Automata describing the application normal behavior.

3) Timed Automata: In addition, a list of acceptable [min, max] duration intervals, including a margin to reduce false positives, is associated to each transition. This list is obtained

[1]One-hot is an array of bits among which the legal combinations of values are only those with a single high (1) bit and all the others low (0)

from the preprocessing phase using k-means clustering algorithm.

E. Testing Phase

For the testing phase, each data vector of the remaining preprocessed data files is evaluated regarding the model previously built. The timestamps of the data vectors considered as anomalies are stored in a list. Then, the maximum number of anomalies in a sliding window of a predefined duration s is computed for each file. If this maximum is higher than a predefined threshold, the file is considered as containing an attack. Else, the file is considered as normal. The evaluation of the data vectors depends on the model used.

1) OCSVM: The detection algorithm computes a distance between the data vector and the OCSVM model (e.g. the hypersurface) to determine if the current data vector is consistent with the OCSVM model (normal) or not (anomaly).

2) Automata: The detection algorithm consists in checking if the current data vector is included inside the list of data vector observed during training. In this case, the boundary is defined as 0 because no new sequence should be allowed.

3) Timed Automata: The detection algorithm consists in checking if the current data vector is included inside the model. It means that the sequence has to exist in the model, and that the duration should be included in one of the corresponding intervals.

IV. Evaluation of the HIDS Solutions Proposed

This section presents the implementation of the process described in Section III to evaluate the detection accuracy of the proposed HIDS solutions (see Table II).

A. Experimental Environment

We have considered two avionic applications in our experiments, namely HSIV (Human-System Interface Vehicle) and CrewB. They are used to display information to the pilots through the cockpit screen. HSIV displays information about the aircraft itself, such as hydraulic systems, fuel, or engine, while CrewB information concerns the flight, such as airspeed, altitude, roll. The applications are executed according to a predefined scenario, without any human interaction, on a real avionic computer (target). They both exhibit a periodic behavior after the initialization phase, without reaching any specific error state.

The prototype illustrated in Figure 4 is used to collect raw data from an application (example with HSIV application on Figure 4). Raw data are extracted using a GNU debugger (GDB): a GDB client runs on the controller and communicates with a GDB server running on the target. An attack injection tool, detailed in [30], is also used on the controller to inject code mutation into the monitored application to emulate malicious behavior. Three mutation code strategies are used in order to emulate a corrupted behavior, by introducing random instructions, replacing sets of instructions, or introducing modifications based on attack patterns. Only one mutation is performed in each experiment. The mutated code can be executed one time or multiple times, depending on its location and its impact on the application.

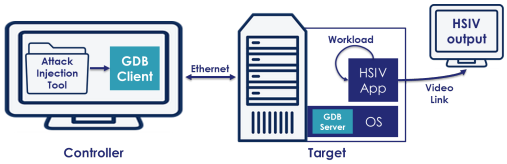

Figure 4: Raw Data Extraction from Application's Execution

Table III: HIDS Solutions Detection Results

Solution	Score on HSIV	Score on CrewB	Mean
OCSVM_all	83.72%	78.18%	80.95%
A_all	27.91%	49.09%	38.50%
TA_all	91.63%	93.94%	92.78%
OCSVM_comms	90%	77.58%	83.79%
TA_comms	93.13%	92.73%	92.93%
OCSVM_API_freq	73.95%	53.33%	63.64%
OCSVM_seq_freq	100%	90.30%	95.15%

The dataset collected for the HSIV application is composed of 39 normal and 43 attack log files, with a duration of 5 to 200 seconds (100 to 4000 execution slots, 4000 to 689000 logs). The dataset for the CrewB application is composed of 30 normal and 34 attack log files, captured during 10 to 30 seconds (200 to 600 execution slots, 64000 to 215000 logs). The environment is reset at the end of each execution in order to prevent any side effect between two consecutive executions.

B. Detection Results

Table III reports the detection scores obtained with the different HIDS solutions using the raw data log files collected from the HSIV and CrewB applications. For both applications, five different training files, with different duration, have been used. For each training file, many experiments have been executed in order to optimize the score (between 4 and 36000 experiments, depending on the HIDS solution evaluated). In each experiment, a different set of parameters is applied (e.g. the length of the sequence, the OCSVM parameters, the margins, the testing boundary). The sequence length is the only parameter of the AS_all solution (leading to 4 experiments only), while many combination of parameters are applicable for the OCSVM-related solutions (leading to 36000 experiments). The resulting score is the mean of the best scores obtained for each training file.

The "OCSVM_seq_freq" solution exhibits the best detection results, especially on the HSIV application, for which a 100% detection score is obtained, regardless of the training set used. Timed Automata anomaly detection techniques ("TA_all" and "TA_comms") also give very good results, on both applications. Indeed, every normal file is detected as normal with a high accurate detection of attack files.

The analysis of the experiments for a given sequence length showed that the size of the sequence does not have a significant impact on the results. As a consequence, the smallest size can be used to reduce the resources needed to process the

Table IV: HSIV and CrewB Applications Characteristics

	HSIV	CrewB
Number of API calls performed per execution slot	83	173
Number of communication-related API calls performed per execution slot	28 (33.7%)	143 (82.1%)
Number of different API calls	23	30
Number of different API calls sequences	all(comms)	all(comms)
sequence size = 2	31(9)	43(25)
sequence size = 3	42(13)	90(56)
sequence size = 4	52(17)	124(88)
sequence size = 5	62(21)	145(106)

data (memory and CPU), which is important in our specific avionics context. The duration of the training phase has an impact on the results for the CrewB application. Indeed, the smallest training set (10 seconds) was too small to capture the entire behavior of the application and gave the worst results. Also, the tuning of the parameters was harder on the OCSVM-based solution than for the others, and the evaluation process was generally very time-consuming.

V. Resources consumption

The three HIDS solutions, "OCSVM_seq_freq", "TA_all" and "TA_comms", exhibit similar results in terms of detection accuracy and can be good candidates to be embedded in an aircraft. Nevertheless, the resource consumption of each solution must be assessed to ensure that it is consistent with the limited resources available in an avionics context. For that purpose, we have assessed the following performance overhead of each solution: 1) Monitoring overhead (O1 & O2), 2) CPU consumption (O3), and 3) Size of the logs (O4).

These first analyses have been performed offline on a classical desktop computer. The three selected solutions have been implemented in python and run on an Intel® Core™ i7-6820HQ CPU @ 2.70 GHz × 8 computer with 16 GiB of RAM. The resource consumption estimated are not necessarily representative of a real embedded HIDS, but they allow a preliminary comparison between the three selected solutions. Table IV presents some characteristics of the HSIV and CrewB applications, observed from the datasets. These characteristics are used to estimate the real-time impact, CPU consumption, or memory space of the HIDS. In order to be representative of the final embedded implementation, the data formatting of the "OCSVM_seq_freq" preprocessing is directly realized during the monitoring. The remaining preprocessing tasks only are considered to evaluate its CPU consumption.

A. Monitoring overhead

The estimation of the monitoring overhead is based on the complexity of the operations performed by the OS each time an API call is made. It is estimated as follows:

- TA_all: For each API call, the corresponding ID and the current timestamp are logged (2 operations * Number of API calls)
- TA_comms: For each communication API call, the corresponding ID and the current timestamp are logged (2 operations * Number of communication-related API calls)
- OCSVM_seq_freq: For each API call, the current sequence and its duration are computed and the corresponding counter, the min, max and mean duration are updated (around 10 operations * Number of API calls)

Obviously, the "OCSVM_seq_freq" solution suffers from a higher overhead than the "TA_all" and "TA_comms" solutions.

Table V: Average CPU consumption per execution slot

Solution	HSIV	CrewB
TA_all	83 µs	182 µs
TA_comms	32 µs	138 µs
OCSVM_seq_freq	76 µs	77 µs

B. CPU consumption

The CPU consumption corresponds to the duration of the data preprocessing phase and the detection phase for one execution slot. For each API call, the data preprocessing and detection phases for "TA_all" and "TA_comms solutions" consist in 1) computing the current sequence and duration and 2) checking if it is included in the automata model. For the "OCSVM_seq_freq", the data preprocessing phase consists in scaling the feature vector and the detection phase consists in executing the OCSVM classifier. Using the traces generated from 500 execution slots, a mean duration per execution slot for each solution were computed. These measures are summarized in Table V.

For the HSIV application, "TA_comms" is the most CPU-efficient solution, while "TA_all" and "OCSVM_seq_freq" presents similar results. The "OCSVM_seq_freq" becomes better than "TA_all" and "TA_comms" for the CrewB application which performs more API calls. Let us note that this "OCSVM_seq_freq" solution exhibits a constant CPU consumption for preprocessing and detection phase regardless of the number of API calls considered and that this solution becomes the most interesting for applications making many API calls.

C. Size of the logs

The size of the logs has been estimated as follows:

- TA_all : 5 bytes (1 for the API call ID, 4 for the timestamp) for each API call performed
- TA_comms : 5 bytes (1 for the API call ID, 4 for the timestamp) for each communication API call
- OCSVM_seq_freq : 4 bytes for the counter plus 3*4 bytes for the min, max and mean duration, per number of different API call sequences (see Table IV)

Figure 5 shows the estimated size in bytes of the generated logs for HSIV and CrewB applications, according to the number of API calls executed. The gray vertical lines show the number of API calls actually performed by the application (HSIV or CrewB) during the experiment.

For the HSIV application, the "TA_comms" solution exhibits the best solution in terms of log size, while the "TA_all"

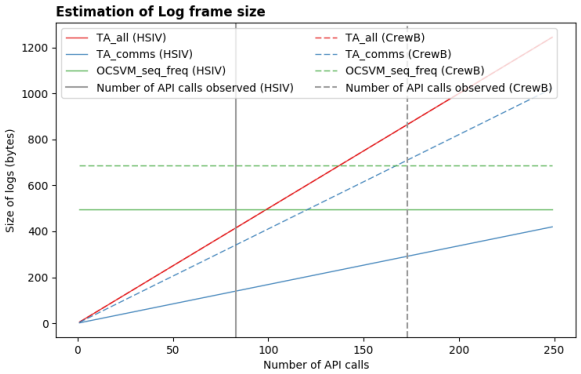

Figure 5: Estimated Log Frame size per API call

and "OCSVM_seq_freq" have similar log sizes if we consider the actual number of API calls performed by this application. Concerning the CrewB application, the three solutions are similar. However, the fact that the log size associated to the "OCSVM_seq_freq" model is bounded makes it a better solution for more complex applications.

D. Conclusion

The evaluation of the monitoring overhead shows that the "TA_comms" solution provides very good results for the two applications studied. Regarding the CPU consumption and the size of the logs, the "TA_comms" and "OCSVM_seq_freq" provide similar results. The "OCSVM_seq_freq" may probably be a better solution for more complex applications performing many API calls during each execution slot, due to the fact that this solution uses a fixed-size log at each execution slot, allowing for a controlled detection time and memory consumption. However, a Timed Automata seems easier to interpret than an OCSVM model (objective O5), and more particularly, the "TA_comms" solution exhibits the smallest impact on the application under monitoring (objective O1).

In the next section, we describe the implementation of the "TA_comms" solution on a real avionic platform, as well as some performance evaluation results.

VI. Embedded HIDS Implementation

The embedded prototype is based on an IMA architecture. The HIDS is implemented within a dedicated avionic partition. This choice allows to limit the impact on the other partitions to the monitoring overhead thanks to the IMA architecture properties (objective O1). Also, the HIDS can be easily integrated into existing legacy aircraft architectures with minimal evolution required (objective O2).

The monitored application is an advanced version of the CrewB application and has slightly different features from the previous version. The following section presents the approach to performing real-time detection (Section VI-A), the different components of the embedded HIDS prototype developed (Section VI-B), and the evaluation of its performance (Section VI-C).

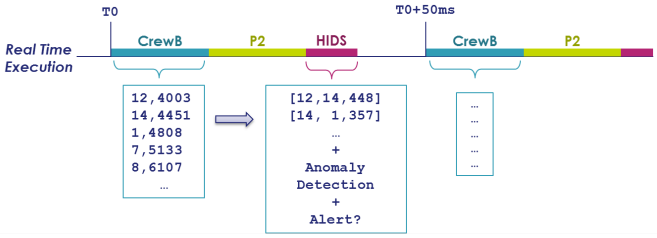

Figure 6: Real-time Execution Configuration

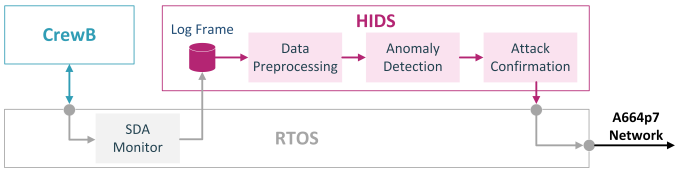

Figure 7: Architecture of the Embedded Prototype

A. Real-time Execution

The processor used in this prototype is a T2081 1.8 GHz. Its use is restricted to a single core, and a static configuration determines the CPU resources allocated to each partition. Each partition is statically scheduled inside a 50 ms execution slot. The monitored application (CrewB) is composed of one partition which runs during 6.7 ms at each period, and the HIDS partition runs during 3.8 ms.

Figure 6 illustrates the real-time execution configuration of the prototype. During the execution slot of CrewB, the RTOS intercepts each API call performed by the CrewB partition and logs the corresponding ID and timestamp in a dedicated memory area within the HIDS partition. Other partitions may be executed between CrewB and HIDS execution slots (for example, "P2" in Figure 6). When the HIDS partition is executed, it preprocesses the logs gathered from the CrewB partition (to aggregate the API call IDs in sequences and to compute the duration of each sequence), executes the anomaly detection algorithm, and sends an alert if necessary. The logs are flushed at the end of the HIDS execution slot.

B. Architecture

The embedded architecture is presented Figure 7. The "SDA Monitor" component, introduced inside the RTOS, intercepts each API call performed by the CrewB application and writes a log in a dedicated memory area of the HIDS partition called "Log Frame". Three components are implemented inside the HIDS partition: the "Data Preprocessing", the "Anomaly Detection", and the "Attack Confirmation". If an alert is raised by the "Attack Confirmation" component, a message is sent on the network (ARINC 664p7).

1) SDA Monitor: The SDA Monitor is an additional assembly code inserted inside the CrewB application. It is introduced at the start of the module using a GDB script. This solution is easier to implement than a direct modification of the RTOS, while remaining representative of the overhead introduced by the addition of monitoring. In a final implementation, the

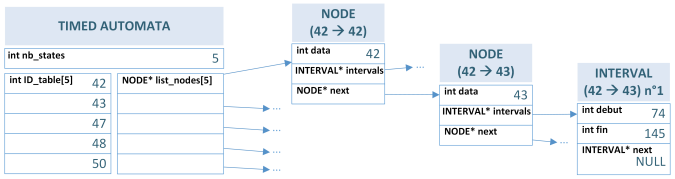

Figure 8: Timed Automata Implementation

monitoring should be done directly by the RTOS. The total size of the instrumentation code is 140 bytes (35 instructions, each instruction is 32-bits long).

2) Data Preprocessing & Anomaly Detection: The HIDS partition performs the preprocessing and anomaly detection on each log within the Log Frame. The Timed Automata model implementation is illustrated in Figure 8. The preprocessing consists in retrieving the *input_index* (from the ID of the log and the ID_Table of the model), the *output_ID* (from the ID of the next log), and the *duration* of the current sequence (from the timestamps of the current log and next log). The anomaly detection consists in checking if the model contains a particular *(input_index, output_ID, duration)* triplet. This is done by 1) searching if the node *[input_index,output_ID]* exists, and 2) searching if one of the intervals of this node contains *duration*, as illustrated in the following code:

```
log1 = (42,100)
log2 = (43,200)
# input_index=0, output_ID=43, duration=100
(input_index, output_ID, duration) =
↪  preprocessing(log1,log2)
# Anomaly Detection
# 1. Does a node [42->43] exist ?
current_node =
↪  search(list_nodes[input_index],output_ID)
if current_node==NULL:
  raise_anomaly(log2.timestamp)
else:
  # 2. Is 100 within the interval of node [42->43]?
  current_interval =
  ↪  search(current_node.intervals,duration)
  if current_interval==NULL:
    raise_anomaly(log2.timestamp)
```

3) Attack Confirmation: Each anomaly timestamp is stored in a fixed-size buffer. The size of the buffer corresponds to the tolerated number of anomalies raised during 0.5 s. A buffer size of 20 means that a normal execution should not raise more than 20 anomalies during 0.5 s. A structure manages this buffer through the indexes of the first and last anomalies stored. If the buffer is full, an alert message is sent via a SAMPLING_PORT API 653 service with the timestamp of the last anomaly. At the end of the HIDS execution, the buffer is updated to remove obsolete anomalies.

C. Performances Evaluation

The monitoring overhead and the detection duration were evaluated for this embedded prototype. To obtain the monitoring overhead, two breakpoints are placed at the entry point and at the final instruction of the syscall library and log the value

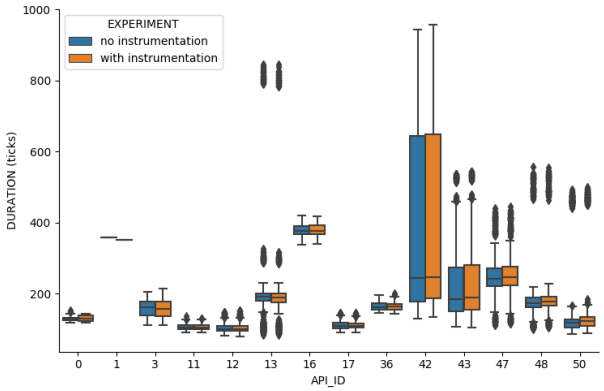

Figure 9: Duration of API calls, w/ and w/o instrumentation

of the TBL register. The *TBL* register contains the value of a 32-bits timestamp based on the frequency of an internal bus (1 tick = 1/37500 ms). Two sets of logs have been captured: one without instrumentation, and one with instrumentation. The corresponding distribution of the duration is shown by Figure 9. An additional overhead is added to the communication services (ID 42, 43, 47, 48 and 50), but remains very low. The median variation is 5 ticks = 135 ns = 2.7%. There is no significant impact on the other services.

Regarding the HIDS execution time, the same principle is used (two breakpoints giving the starting and ending timestamp). For an average of 140 API calls performed at each execution slot (indeed, this is a new version of the CrewB application and its behavior is slightly different), the mean execution duration of the HIDS is 524 ticks $= 0.014\,ms$. This means that the HIDS is able to process the API calls carried out by the application during its execution slot (6.7 ms) in only 0.014 ms (0.21%).

However, in the worst case theoretical scenario, an application could loop on the same API call (i.e., waiting for an answer to a request). In this case, we consider that the minimum duration between two API calls is 50 ticks (as observed in Figure 9). The maximum number of logs would be 5025, and the estimated time to compute these 5025 logs would be $5025 \times 524/140$ = 18.864 ticks = 0.50 ms (7.5%). This result remains acceptable, but must still be validated by a dedicated experiment.

VII. Conclusion and Future Work

In this paper, we proposed the design and the implementation of an HIDS aiming at detecting malicious applications embedded into an aircraft. Six context-specific objectives have been defined in order to develop such HIDS. These objectives are related to the impact of the HIDS on the execution environment (other applications, legacy aircraft), the use of resources (CPU consumption, memory footprint), the exploitation of results, and the possibility of setting up the HIDS for "black-box" applications. The proposed approach consists in building a model of the legitimate behavior of the application monitored, during the integration phase, and

detecting deviations from this behavior, during the operation phase. The model of the legitimate behavior is learned from the sequence of ARINC 653 API calls performed by the application as well as their duration, by using either automata, timed automata or OCSVM classifier.

Some factors may limit the scope of our results. Even if the experimented applications have a simple and periodic behavior, similar applications are actually embedded in the aircraft (for example, to collect information from the aircraft sensors and provide them to other applications). Also, we experimented with a small number of attack performed for both applications, too few to provide a satisfactory detection analysis per attack class.

However, the use of a Timed Automata to model the communication-related API calls showed very good detection results and interesting properties to be embedded in the given context. This solution has been implemented inside an avionic computer to be evaluated on a real dedicated hardware. The experimental result show that the HIDS can perform the real-time detection (0.014 ms to process the logs generated by an application with an allocated execution slot of 6.7 ms) while introducing a very small monitoring overhead at each API call performed by the application (2.7%).

In further work, we plan to verify if this HIDS is able to monitor multiple applications. It would be interesting to analyze to what extent such an HIDS could perform efficiently and to identify its limitations (for example, the number of applications that can be monitored simultaneously). We also plan to go further in the definition of the approach by analyzing the interesting data to provide when an alert is raised, such as registers value or stack values, in order to help a security analyst to perform a diagnosis.

REFERENCES

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

[2] S. Parkinson, P. Ward, K. Wilson, and J. Miller, "Cyber Threats Facing Autonomous and Connected Vehicles: Future Challenges," *IEEE Transactions on Intelligent Transportation Systems*, vol. 18, no. 11, pp. 2898–2915, Nov. 2017. [Online]. Available: http://ieeexplore.ieee.org/document/7872388/

[3] C. Biesecker, "Boeing 757 testing shows airplanes vulnerable to hacking, dhs says," *Aviation Today*, Nov. 2017.

[4] "ED-202/DO-326: Airworthiness Security Process Specification," *EUROCAE WG-72 and RTCA SC-216*, Oct. 2010.

[5] "ED-203/DO-356: Airworthiness Security Methods and Considerations," *EUROCAE WG-72 and RTCA SC-216*, Sept. 2015.

[6] "ED-204/DO-355: Information Security Guidance for Continuing Airworthiness," *EUROCAE WG-72 and RTCA SC-216*, June 2014.

[7] A. I. Activities, "Arinc 664 p5: Aircraft data network part 5 network domain characteristics and interconnection," 2005.

[8] M. P. K. Netkachova, K. Müller and R. Bloomfield, "Investigation into a layered approach to architecting security-informed safety cases," *IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*, 2015.

[9] A. A. Dessiatnikoff, Y. Deswarte and V. Nicomette, "Securing integrated modular avionics computers," *32nd Digital Avionics Systems Conference (DASC)*, Oct. 2013.

[10] K. O'Neill, G. R. Newell, and S. K. Odiga, "Protecting flight critical systems against security threats in commercial air transportation," in *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, Sep. 2016, pp. 1–7.

[11] F. M. Tabrizi and K. Pattabiraman, "Flexible Intrusion Detection Systems for Memory-Constrained Embedded Systems," in *Dependable Computing Conference (EDCC), 2015 Eleventh European*. IEEE, 2015.

[12] I. Studnia, E. Alata, V. Nicomette, M. Kaâniche, and Y. Laarouchi, "A language-based intrusion detection approach for automotive embedded networks," *21st IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2015), Zhangjiajie, China*, Nov. 2014.

[13] 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 *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*. IEEE, 2013.

[14] G. Kim, S. Lee, and S. Kim, "A novel hybrid intrusion detection method integrating anomaly detection with misuse detection," *Expert Systems with Applications*, vol. 41, no. 4, pp. 1690–1700, Mar. 2014. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0957417413006878

[15] H. Om and A. Kundu, "A hybrid system for reducing the false alarm rate of anomaly intrusion detection system," in *Recent Advances in Information Technology (RAIT), 2012 1st International Conference on*. IEEE, 2012, pp. 131–136.

[16] S. G. Casals, P. Owezarski, and G. Descargues, "Generic and autonomous system for airborne networks cyber-threat detection," *32nd Digital Avionics Systems Conference (DASC), Syracuse, NY*, Oct. 2013.

[17] P. Parkinson and L. Kinnan, "Safety-critical software development for integrated modular avionics," Wind River, Tech. Rep., 2018. [Online]. Available: http://events.windriver.com/wrcd01/wrcm/2015/02/Safety-Critical-Software-Development-for-Integrated-Modular-Avionics-White-Paper-1.pdf

[18] M.-K. Yoon, L. Sha, S. Mohan, and J. Choi, "Memory heat map: anomaly detection in real-time embedded systems using memory behavior." ACM Press, 2015, pp. 1–6. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2744769.2744869

[19] M.-K. Yoon, S. Mohan, J. Choi, M. Christodorescu, and L. Sha, "Learning Execution Contexts from System Call Distributions for Intrusion Detection in Embedded Systems," *arXiv preprint arXiv:1501.05963*, 2015.

[20] T. R. Glass-Vanderlan, M. D. Iannacone, M. S. Vincent, Q. Chen, and R. A. Bridges, "A survey of intrusion detection systems leveraging host data," *CoRR*, vol. abs/1805.06070, 2018. [Online]. Available: http://arxiv.org/abs/1805.06070

[21] P. J. Prisaznuk, "Arinc 653 role in integrated modular avionics (ima)," *IEEE/AIAA 27th Digital Avionics Systems Conference (DASC)*, 2008.

[22] D. Kwon, H. Kim, J. Kim, S. C. Suh, I. Kim, and K. J. Kim, "A survey of deep learning-based network anomaly detection," *Cluster Computing*, Sep. 2017. [Online]. Available: http://link.springer.com/10.1007/s10586-017-1117-8

[23] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Computing Surveys*, vol. 41, no. 3, pp. 1–58, Jul. 2009. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1541880.1541882

[24] A. L. Buczak and E. Guven, "A Survey of Data Mining and Machine Learning Methods for Cyber Security Intrusion Detection," *IEEE Communications Surveys Tutorials*, vol. 18, no. 2, pp. 1153–1176, 2016.

[25] L. A. Maglaras, J. Jiang, and T. J. Cruz, "Combining ensemble methods and social network metrics for improving accuracy of OCSVM on intrusion detection in SCADA systems," *Journal of Information Security and Applications*, vol. 30, pp. 15–26, Oct. 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2214212616300229

[26] B. L. Daley, "USBeSafe: Applying One Class SVM for Effective USB Event Anomaly Detection," Northeastern University, College ofComputer and Information Systems Boston United States, Tech. Rep., 2016.

[27] X. Liu, Q. Lin, S. Verwer, and D. Jarnikov, "Anomaly Detection in a Digital Video Broadcasting System Using Timed Automata," *arXiv:1705.09650 [cs]*, May 2017, arXiv: 1705.09650. [Online]. Available: http://arxiv.org/abs/1705.09650

[28] T. Klerx, M. Anderka, H. K. Büning, and S. Priesterjahn, "Model-Based Anomaly Detection for Discrete Event Systems," in *2014 IEEE 26th International Conference on Tools with Artificial Intelligence*, Nov. 2014.

[29] B. Scholkopf, R. Williamson, A. Smola, J. Shawe-Taylor, and J. Platt, "Support Vector Method for Novelty Detection," p. 7, 2001.

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