Anomaly based Intrusion Detection for an Avionic Embedded System

Aliénor Damien, Marc Fumey, Eric Alata, Mohamed Kaâniche, Vincent Nicomette

To cite this version:

Anomaly based Intrusion Detection for an Avionic Embedded System

Aliénor DAMIEN[1,2], Marc FUMEY[1], Eric ALATA[2,3], Mohamed KÂANICHE[2], Vincent NICOMETTE[2,3]

[1]Thales AVS France, [2]LAAS-CNRS, Université de Toulouse, CNRS, [3]INSA

Abstract

This paper firstly describes the challenges raised by the introduction of Intrusion Detection Systems (IDS) in avionic systems. In particular, we discuss some specific characteristics of such systems and the advantages and limitations of signature-based and anomaly-based techniques in an avionics context. Based on this analysis, a framework is proposed to integrate a Host-based Intrusion Detection System (HIDS) in the general Integrated Modular Avionics (IMA) development process, which fits avionic systems constraints. The proposed HIDS architecture is composed of three modules: anomaly detection, attack confirmation, and alert sending. To demonstrate the efficiency of this HIDS, an attack injection module has also been developed. The overall approach is implemented on an IMA platform running a cockpit display function, to be representative of embedded avionic systems.

I. Introduction

The threat surface of an aircraft has always been very reduced thanks to strong safety requirements [1], [2], design isolation of critical cockpit functions, and limited connectivity. The fault-tolerant hardware platforms were mainly dedicated to avionics domain, with specific operating systems, preventing them from standard malware attacks. However, the trend nowadays is to make aircraft systems connected and less expensive, for example by providing the capacity to update the weather data in flight or by drastically increasing the number of functions of mixed criticality levels on common platforms [3]. These evolutions increase the threat surface that an attacker could potentially use to compromise the system.

Moreover, the attack techniques are evolving and even specific and isolated systems can be hacked [4]. Security research teams are also working on the security of embedded systems like cars [5] and aircraft [6], [7], highlighting the need for security measures on these systems. Even if the avionics domain remains one of the safest up-to-now, these examples show that actual certification process and safety mechanisms may not be sufficient to protect against deliberate attacks in the future.

There are some fundamental differences between accidental faults and deliberate attacks. Because accidental fault occurrences are not deliberate, system safety requirements are mostly based on the probability of fault occurrences, and protection mechanisms are provided to prevent single fault conditions to lead to catastrophic consequences.

As a consequence, the probability of having combined accidental faults is considered low. However, from a security point of view, an attacker is likely to carry out correlated malicious actions. Attacks become more and more sophisticated, may use 0-day exploits (e.g. exploitation of non-yet patched vulnerabilities), and target a specific goal. In this way, an attack may cause many correlated errors to modify slightly the execution flow or a critical data to affect the integrity or availability of an avionic function. Even if, thanks to architectural properties such as segregation, redundancy, dissimilarity, it remains very hard to conduct an attack with a catastrophic impact on an avionic function, we argue that some attacks may not be covered by safety mechanisms and that some additional mechanisms must be included for that purpose.

This paper presents an ongoing work aimed at developing an Intrusion Detection System (IDS) for Integrated Modular Avionics (IMA) platforms, considering the point of view of a module integrator. The presentation is focused on the detection architecture and the detailed description of its components. The detection of a security event may consist of 1) logging the alerts for a posteriori investigation and/or 2) reacting automatically on-board to recover from faults generated by the attack. This paper only addresses the first point and does not investigate how the system should react when an intrusion is detected. Section II describes the context and the scope of this study. Section III focuses on existing security mechanisms deployed in avionics domain and on existing IDS used in traditional information systems, and analyzes their advantages and limitations with respect to the specific constraints inherent to avionic systems. Then Section IV presents our approach to integrate a Host-based IDS (HIDS) in an avionics context. A prototype of this approach is presented in Section V. Some preliminary results are presented in Section VI. Section VII concludes and discusses future work.

II. Context

This section presents the different actors involved in an Integrated Modular Avionics process, their interactions, and the threat model considered in this paper.

A. Actors involved

Many actors are involved in a conventional Integrated Modular Avionics (IMA) process, with different roles, responsibilities, and interactions between each others: module suppliers, application suppliers, module integrator, system and aircraft integrators, and airline company. Module suppliers provide a hardware platform with a Real-Time Operating System (RTOS) and associated tools. Application suppliers are mainly involved in the avionic functional software development phase. The module integrator follows the progress of the application suppliers during development and is responsible for the applications integration on the IMA platform. System and Aircraft integrators are responsible for functional integration. Airline is only involved in the operation phase.

Table 1 summarizes these roles and phases.

Table 1. Roles of Actors Involved in Conventional Avionic Process

Actor	Phase	Role
Module supplier	1. Module Development	Develop hardware platform, RTOS and integration toolset
Application suppliers	2. Application Development	Develop an application to provide an avionic functionality
Module integrator	2. Application Development	Allocate resources to the different applications
	3. Integration	Install applications on a module and perform the certification of the whole
System and Aircraft	3. Integration	Integration & Functional integration of the whole system
Airline	4. Operation	Operate and maintain the aircraft

Figure 1 presents the main interactions between the actors of the IMA process. The module integrator dispatches a set of available resources between the application suppliers using a contract-based approach [8]. The ARINC 653 standard [9] proposes an Application Programming Interface (API) that allows the application suppliers to develop their applications by only knowing the resources they can use. The software development process, functional tests, and binary generation (also called "loadable" in avionics domain) are managed by each application supplier. The application loadables only are sent to the module integrator whose role is to integrate them into the whole system, composed of the hardware module and RTOS provided by the module supplier, the resources configuration generated from the resources allocated to each application, and each application loadable. After the platform system and aircraft integration phases, the data loading of each loadable on the aircraft is performed by the maintenance operators of the airline, potentially all around the world.

From a security point of view, the role of the module integrator is very important. He has to ensure a correct resource sharing between applications, validating in particular the spatial and temporal segregation between applications potentially with different criticality levels delivered by all application suppliers. He should then ensure that any application does not exhibit undesired interactions with other applications. In this paper, we adopt the module integrator point of view to address security.

A difficulty in such context is the number of different stake holders potentially from various companies involved, implying the confidentiality of application's documentation. Moreover, it is quite unusual, and very unlikely that the module integrator needs to have access to the specification and functional data of the applications he has to integrate. Nevertheless, he has to guarantee the spatial and temporal segregation between the applications running on the platform. In the rest of this paper, activities presented as Module Integrator activities may in fact be distributed between the Module Integrator and the System/Aircraft Integrator depending on the design of the integration process.
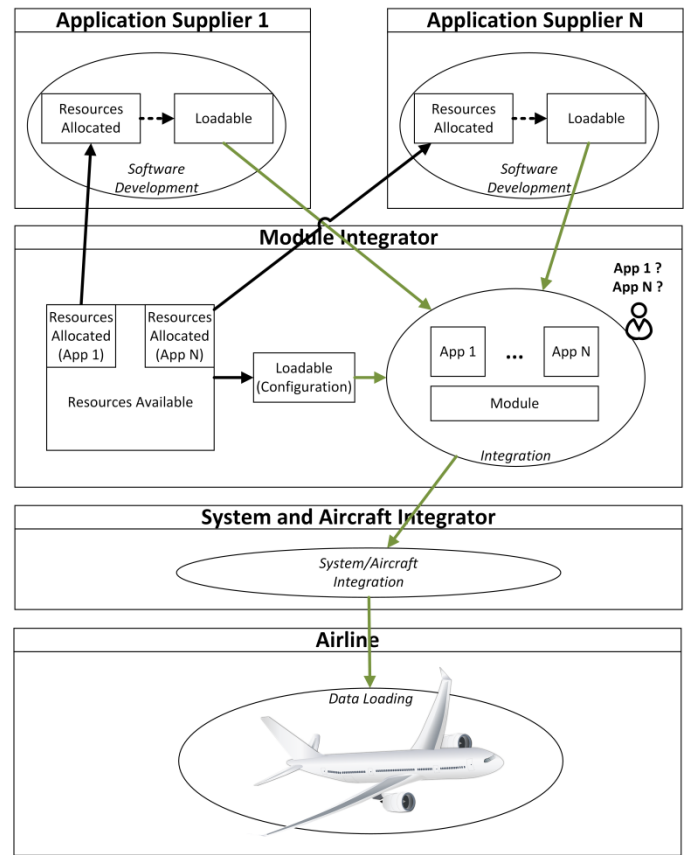

Figure 1. Actors Interactions in an IMA Process

B. Threat Model

Due to the complexity and the criticality of the domain, each actor has a precise role in the certification process of an aircraft. The application suppliers have to provide guarantees about each functionality they implement (intended function). The module supplier provides guarantees about the capacities of his hardware platform, the performances of his RTOS, the relevant constraints for the users (usage domain), and the spatial and temporal segregation properties of his platform. The module integrator has to guarantee that the integration is compliant with both the application requirements and the platform capacities and constraints.

To help the other actors, the module supplier can provide a qualified toolset with a compiler, a configuration tool, a usage domain checker, and a load generator [8]. In particular, the application suppliers and module integrator have to check some configuration rules to be allowed to generate the loads. The qualified tool chain provides the capability for each actor to generate loads that can be installed inside an aircraft. Of course, there are mechanisms to check the integrity of the loads, which are designed for safety purpose. There are also mechanisms to authenticate the load (signature-based loads).

This hypothesis of mutual trust cannot be made when considering security purposes. There are only a few companies that provide modules or manage integration, and these companies can be trusted. In the opposite way, there are many different application suppliers and airlines all around the world, and we consider here that they cannot be trusted. In particular, this makes the application development and maintenance phases more critical.

Another threat is due to the installation of partitions with multiple criticality levels on a same module. Considering the high safety requirements for Design Assurance Level (DAL) A applications [2], it should be very hard to find a vulnerability inside this type of application. However, less critical functions are verified with a lower level of rigor, and could be corrupted in order to perform attacks on a more critical function through allowed interfaces.

Platforms supporting these applications are developed at the highest criticality compared to applications (typically DAL A in avionics context). These platforms implement a robust partitioning to protect the platform and other hosted applications from any misbehavior of any of these applications. An attack from an application impacting other applications seems then very unlikely on an IMA platform. Nevertheless, some recent examples suggest attacks are still possible through hardware vulnerabilities [10].

These limitations raise two challenges to secure such systems:

- If a binary is malevolent, it has to be detected before it is embedded,
- If a binary is vulnerable, its possible corruption has to be detected at the runtime.

In this paper, we propose to adapt the IMA process and tools, to detect such malevolent or compromised application during the integration phase or at runtime.

III.   Avionics Security State of the Art

This section presents the existing approaches to integrate security in the avionic systems, and more precisely some studies about Intrusion Detection Systems (IDS) in embedded systems.

A.   Security in Critical Avionic Systems

Considering some recent attacks on embedded systems [11], [6], the navigability regulation has evolved [12], [13]. Nowadays, avionics actors have to consider on-board and ground infrastructure security. This evolution has been included in the design of new aircraft [14]. Perimetric defenses are implemented to split the network into different domains [15], [16]. Systematic vulnerability analysis during the development of new platforms is also now considered, as presented for instance in [17]. 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 [18].

However, to the best of our knowledge, they do not implement yet Intrusion Detection Systems (IDS) in operation. 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.

B.   IDS in Embedded Systems

IDS are usually classified as Signature-based and Anomaly-based IDS [19]. The first ones look for attack patterns, and the other ones for deviations from a normal behavior. To apply these techniques to avionic systems, the embedded IDS to be designed has to take into account the following constraints:

- Real-time: It must not disturb the real-time execution of the aircraft functions.
- High safety level: It should not directly affect the aircraft safety or introduce new dependencies between applications.
- Performance: It must not consume too many resources.
- Maintenance: It should not be updated at each landing to take into account recently discovered vulnerabilities because of the high cost of a grounded aircraft for the airline.
- Life time: It must be efficient during at least 20 years.
- Certification: It must satisfy the Development Assurance Level (DAL) [2] requirements assigned to the IDS.
- Resilience to attacks: It should be protected against malicious potential corruption.

Signature-based IDS need a database of known attacks (that does not exist today for avionic systems) with frequent updates. They are not suitable to detect new or sophisticated attacks. As a consequence, this kind of IDS is not suitable for the avionics context. On the other hand, anomaly-based IDS may generate a lot of false alarms, and the cost of grounding a fleet due to a false alert is not acceptable. Moreover, some difficulties may be raised with respect to certification if the algorithms implemented in the IDS are not deterministic. However, anomaly-based IDS present some interesting characteristics in avionics context. They are efficient to detect new attacks without requiring the update of an attack signature database [20]. Moreover, the modeling of the normal behavior of an avionic application can be performed with a good accuracy because the avionics environment is under strict control and is designed to be deterministic.

A few studies have been published about IDS in embedded critical systems. For instance, [21] highlight some related constraints and challenges. [22] proposes an IDS for embedded automotive architectures. The use and implementation of IDS on multi-core architectures for real-time embedded systems is investigated in [23]. Some studies propose hybrid IDS to take advantage of both signature-based and anomaly-based techniques [24], [25]. Those studies are not specific to avionics domain, except the work of Silvia & al [26] that proposes a network-based IDS using network packets as input data. A limitation of this approach is that it cannot detect attacks that are internal to a module or to an application.

Our research focuses on a different approach, aiming at integrating a Host-based IDS (HIDS) into each module that is designed to monitor the behavior of the hosted applications, using in particular avionic RTOS as a source of data.

Considering both advantages and drawbacks of signature-based and anomaly-based IDS, we propose an hybrid approach with an anomaly detection module and an attack confirmation module based on a knowledge database including attack data as well as known safety-related or false positive data.

IV.   Overall Approach

The proposed HIDS approach aims at covering two types of threats:

- A malevolent loadable sent to the integrator has to be detected during the integration phase. In this case, we assume that the binary only is corrupted, but that the documentation or activation inputs given to the integrator are correct.

- A loadable is corrupted after the integration phase (for example, malicious modification of a loadable on ground, external attack, attack from another corrupted equipment). It has to be detected at runtime.

Figure 2 presents the blocks added during the integration phase and Figure 3 shows the blocks added during the operation phase. They represent on-ground (green) or embedded (blue) tasks.
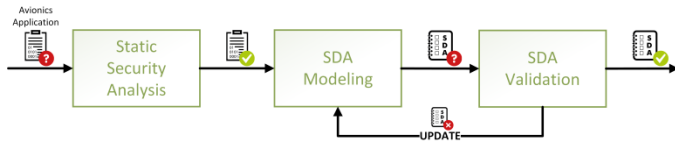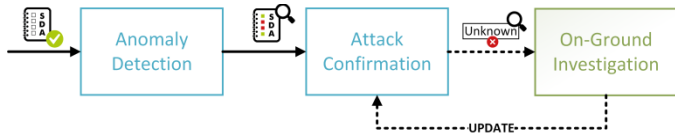

Figure 2. Integration Phase

Figure 3. Operation Phase

The first threat is managed by the "Static Security Analysis" processing block. The validity of the binary received is checked by comparing it with the documentation and existing knowledge about the application.

The second threat is addressed by a hybrid embedded HIDS composed of two embedded blocks: "Anomaly Detection" and "Attack Confirmation". The "Anomaly Detection" block is configured with a "Security Domain of the Application" (SDA), built through the "SDA Modeling" and "SDA Validation" processing blocks during integration phase.

The "Attack Confirmation" block is aimed at reducing the number of false alarms using signatures of already encountered anomalies, stored in a knowledge database. The investigation of unknown anomalies on the ground is represented by the "On-Ground Investigation" processing block. It allows to update the knowledge database of the "Attack Confirmation" block for a whole aircraft fleet after an unknown anomaly is raised by one of them.

The following sections illustrate more precisely the definition of the SDA and the different blocks added to the traditional IMA process.

A. Security Domain of the Application (SDA)

The "Security Domain of the Application" (SDA) is a set of rules characterizing the normal behavior of an application (for example, the application A should not do more than ten API calls in one execution cycle). It is based on the platform resources usage of the application. As an application and its environment are not designed to evolve in real time, it should use the same resources through the lifetime of the aircraft (or until an update).

Its design should be adapted to the targeted platform, taking into account the resources available for the HIDS (in terms of storage or data bandwidth), the information available on the existing platform (like the safety monitoring alerts, the instrumentation means, or the maintenance information), or the possible platform's developments.

The main difficulty for selecting the relevant parameters involved in the definition of the SDA rules is to choose the simplest set of data to monitor, which is the most efficient and well adapted for any kind of application for a given system. Several approaches can be explored to design these parameters:

- Select information that are already available, for example to be compliant with legacy aircraft.
- Select the most critical information based on a risk analysis and an analysis of the effects of attacks on the system.
- Select most common information monitored by HIDS from the literature (in avionics or other domains).
- Select the information to monitor by experiment different HIDS under attack simulation.

Table 2 proposes a first list of high level observation classes to help designing the SDA parameters. Each information monitored should be composed of an observation level and a characteristic to observe.

For example, a SDA could be composed of rules characterizing the number of API calls performed by an application periodically during its execution, the time to perform a sequence of a specific number of API calls, the number of data segment reads and writes or the diversity of instruction types executed.

Table 2. Examples of Observation Classes for SDA Parameters

Observation Levels	
API Call	Raise an event when an API call is made by the application
Code executed	Raise an event when an instruction is executed by the application
Communications	Raise an event when a message is sent or received by the application
CPU Counters	Read the performance monitor registers of the processor
Memory	Raise an event when the application accesses the memory of the platform
OS Errors	Raise an event when an error is raised by the OS
Characteristics	
Diversity	Number of different events
Number	Number of events
Sequence	Sequence of events
Parameters	Parameters of an event
Payload	Payload of an event
Timestamp	Timestamp of an event
Type	Type of an event

B. Static Security Analysis

The Static Security Analysis block aims at detecting a corrupted or malevolent loadable received from an application supplier to be integrated. In this case, we assume that only the loadable is corrupted but not any other documentation received from the application supplier.

Two ideas are pursued to perform this analysis:

- Use existing anti-malware techniques on the binary.

- Check the compliance between the binary and its documentation.

1) Binary Verification with Existing Techniques

It is important to detect the presence of a potential malware embedded inside an application loadable. For example, a corrupted USB key could have been used to transfer the loadable, or the production environment could be corrupted. It is important to investigate the corruption and find its root cause to patch the vulnerability and prevent a targeted malware from using it. The survey published in [27] highlights different approaches used by anti-virus products to detect a known malware. Some other research works also study morphological analysis of malware to recognize known malicious code [28].

2) Binary Compliance with its Documentation

As mentioned in Section II, the module integrator receives the binary to integrate from the application supplier, but he should also handle additional documentation to carry out this verification, like the insertion contract, the source code, the specification, or a description. A static verification of the compliance between the loadable and these documents can be carried out, right after the loadable delivery from the application supplier to the module integrator. This static verification is automated using the Resources Usage Analyzer described in Section V.

In practice, only the insertion contract is known in every case, because it is a formal document written by the module integrator describing the resources allocated to the application supplier. This contract contains information about memory space, CPU time, communication ports, specific services, or partition numbers allocation, but this information is not sufficient to precisely characterize the application's behavior. For example, the contract may stipulate that the application can use 3MB of Non Volatile Memory (NVM), without providing information about its usage. As a consequence, the module integrator does not know if the NVM services are used occasionally, periodically, or very often, a potential NVM flooding cannot be detected using only the information provided by the insertion contract.

This document can be useful to check some security requirements at a high level, but it is not precise enough to perform accurate runtime intrusion detection. The next block (SDA Modeling) aims at extending the insertion contract knowledge to have a more precise model of the application's behavior.

C. SDA Modeling

The objective of this phase is to build a preliminary SDA as described in Section IV-A modeling the normal behavior of the application. It can be done directly from the application's documentation provided in the delivery phase (manually) if it is sufficient. In the most common cases, the SDA is built by running the application in a laboratory setup, by simulating its activation inputs and observing the application resources usage as defined by the SDA parameters. The activation inputs, also provided by the application supplier, must be the same as the inputs used to perform the functional tests on the application. It can be represented by another stimulation partition or a network benchmark. There are two main advantages of reusing the inputs used to perform the functional tests: limiting the development costs and observing the application in all its

modes (as we assume that the functional tests correctly cover all the modes of the application).

Figure 4 illustrates a possible implementation of this block using three modules: an SDA Monitor to collect data about the application, a Data Pre-processing Tool to format these data, and an SDA Learning Tool to learn automatically the application's behavior by using semi-supervised machine learning techniques. In this case, the application is stimulated using a partition that emulates the activation inputs.
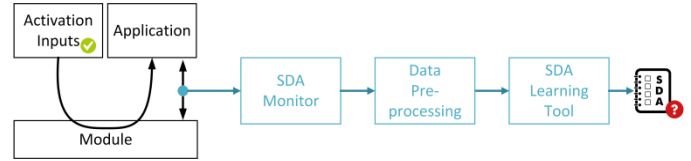

Figure 4. SDA Modeling

D. SDA Validation

In this phase, illustrated by Figure 5, attacks are injected in the application in a lab environment to test the efficiency of the detection. This is an iterative process with the SDA Modeling block, to obtain a more precise SDA. Because we don't have currently any example of compromised application or successful attack, our approach consists in injecting the consequences of a potential attack on an application using an Attack Injection Tool.
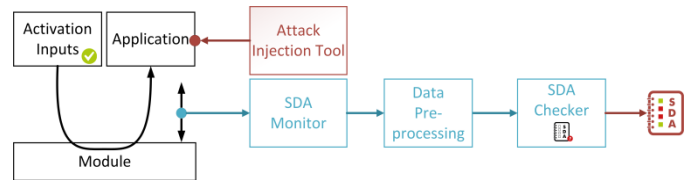

Figure 5. SDA Validation

For example, the attack injected can simulate the deactivation of the error logging process by replacing its code with *nop* instructions. It can also modify an API call by replacing or removing it. Another example consists in the inclusion of an infinite loop inside a process to block the whole partition. More information about the attack injection tool we developed is provided in Section V.

The application behavior under attack is monitored and characterized using the SDA Monitor and the Data Pre-processing Tool presented previously. It is then compared to the SDA obtained previously using the SDA Checker. If the initial SDA was constructed from the application's documentation only, the activation inputs can also be simulated to assess the accuracy of the SDA. If some attacks are not detected or if there are too many false alarms, the results are investigated to propose a new SDA. Finally, the SDA obtained after some iterations is considered to fit the normal correct behavior of the application and each deviation from this behavior is considered as possibly malicious.

E. Anomaly Detection

Once the SDA is validated, it is deployed in the aircraft for the operational phase, as well as the application itself.

The anomaly detection block is embedded on the aircraft, for example within the hardware, the RTOS, or in a dedicated partition. A dedicated system partition (developed with the RTOS and provided into the same loadable) could be a good choice to place the anomaly detection block, as it is compliant to the IMA principles, controlled by a trusted actor, which should facilitate the interface with the RTOS to monitor other applications. Thus, we consider that the anomaly detection block is placed in such partition in this paper.

The role of this block is to monitor the applications resources usage as defined by the SDA parameters, and compare it to the SDA of each application independently. If an anomaly is detected, it is notified to the attack confirmation embedded block described hereafter.

The anomaly detection partition only monitors the application and does not interact with the other partitions. A possible implementation of this monitoring is to modify the RTOS to capture and store the resources usage events into a memory area dedicated to the anomaly detection partition. This solution allows the anomaly detection partition to collect data from the other applications without interacting directly with them. As the partition does not interact with critical applications and has no impact on the aircraft safety, it could have a low DAL. However, the corruption of this partition could be critical from a security point of view, and some measures must be taken to ensure the security of this particular partition.

F. Attack Confirmation and On-Ground Investigation

Even if anomaly detection has many advantages in avionics domain, the rate of false alerts and the potential need to update the model can be challenging. The attack confirmation embedded block is aimed at mitigating this problem by implementing the following functionalities:

- *Anomalies characterization:* It uses a knowledge database to confirm a real attack or to exclude known false positives or safety-related anomalies that are handled by other components of the architecture.
- *Alerts sending:* It sends alerts to the crew and/or to the ground depending on the anomalies characterization results.
- *Knowledge database updating:* It updates the knowledge database after investigation of non-confirmed anomalies on the ground.

This block could be embedded inside a dedicated partition, a dedicated module, or even implemented on-ground depending on its role and the resources available. As the HIDS is designed only for detection and not for automatic reactions as for now, and has to be used for a fleet of aircraft, both online and offline detection might be considered.

Online detection could allow in the future to react very quickly and even automatically. However, it may consume a lot of computational resources on-board and certification concerns can be raised, for example on the crew's ability to react in case of an alert. An offline detection could be a first step to introduce such technique inside an aircraft, and to evaluate its efficiency in operation without interfering with the crew, but this requires a significant amount of storage to save each detection information raised during the flight. In this paper, we chose to implement the offline detection option.

As this block is generic and independent of the avionic application monitored, it is also much easier to update any of its components from a certification point of view. Indeed, the detection application is not critical for the safety of the aircraft.

V.   Prototype Description

To assess the relevance of the overall approach, a first prototype has been developed. Its components, presented in the previous section and summarized in Figure 6, are described in this section:

- The Resources Usage Analyzer is used to statically check the consistency between the resources usage of the application and the resources allocated to it.
- The SDA Monitor captures information about the application's usage of resources while running on the platform.
- The Data Pre-processing Tool is used to format the data received from the SDA Monitor.
- The SDA Learning Tool uses the pre-processed data to model the application's behavior and generate the SDA.
- The Attack Injection Tool emulates attacks on a running application.
- The SDA Checker evaluates the data received from the SDA Monitor and identifies anomalies in the application's current behavior, by comparing it to its SDA.
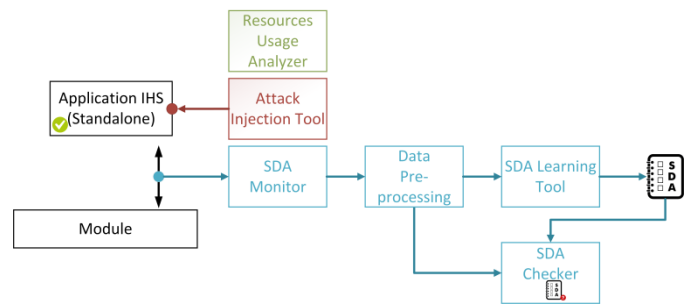

Figure 6. Prototype Architecture

A. Resources Usage Analyzer

The Resources Usage Analyzer aims at computing the metrics characterizing the usage of each type of resources used in the application's code. This is realistic because every resource is allocated statically. The tool is based on the standardized API ARINC653 [9], which classifies the API services provided in different categories described in Table 3. To compute the amount of resources used for a specific family of services, the API calls used to allocate this kind of resource have to be known.

For example, the Non Volatile Memory (NVM) is allocated using the *CREATE_LOGBOOK* and *CREATE_NOTEPAD* API calls. To compute the global allocated memory, the tool parses the code to find where these API calls are used and the values of the parameters that declare the total size to allocate. This information is extracted by parsing the application's code to find function calls. Currently, the prototype is developed for a 32-bits PowerPC architecture where each instruction is coded on 32 bits. Each function call is built with the same structure. The avionic systems environment being static and deterministic, the services are always called with static parameters as input.

Table 3. Categories of API ARINC 653 Services

Partition Management
Process Management
Time Management
Inter-partition communication
Intra-partition communication
Error handling

B. SDA Monitor

The SDA Monitor is used to monitor the application resources usage during its execution. In practice, in order to have full access to the information of the existing platform, the debugging mode (see Section IV-D) is used. Currently, our prototype only monitors the API calls performed by the applications. The principle is to insert a breakpoint at each API call, and to execute a set of logging instructions when the breakpoint is reached. For instance, the following debugging script is aimed at logging the usage and the corresponding date of the *CREATE_PROCESS* API call.

```
b CREATE_PROCESS
commands
    silent
    printf "1,%i,%i\n",  $tbu, $tbl
    c
end
```

These execution traces are collected in the GDB logging file in the following format:

```
12,3117,4003
14,3117,4451
1,3117,4808
7,3117,5133
8,3117,6107
```

The first number corresponds to the ID, the second one to the upper part of the RTOS clock, and the third one to the lower part of the RTOS clock.

C. Data Pre-processing Tool

This tool is dedicated to format the data received from the SDA Monitor. Currently, as our prototype only monitors API calls, this formatting is done in two steps:

- Reconstruct the timestamp and transform the original data file into a list,
- Aggregate the data as API sequences with the duration from the first to the last API call in the sequence.

The timestamp is reconstructed by concatenating the second and the third data from each line of the file. After this first step, the resulting data is a list of couples [ID, Absolute Timestamp].

This list is then aggregated into sequences of a given length. For instance, if we consider sequences of four API calls, the resulting

data are represented by [ID1, ID2, ID3, ID4, duration]. The final dataset is constructed with a sliding window.

If we consider the original dataset presented in Section V-B, the resulting data with sequences of length four is:

```
[12, 14, 1, 7, 1130]
[14, 1, 7, 8, 1656]
```

It can also be represented in two dimensions by concatenating the IDs:

```
[12140107, 1130]
[14010708, 1656]
```

D. SDA Learning Tool

This tool is used to model the application's behavior based on the formatted data, using semi-supervised machine learning techniques. Indeed, machine learning algorithms can provide efficient results to detect anomalies in observations, by means of classification algorithms. The problem we address is related to a one-class classifier that takes as input only elements of one class. Indeed, the dataset collected during the elaboration of the SDA only characterizes the normal behavior of the application. We do not have any dataset of attacks. A one-class classifier is able to build a model fitting the normal behavior of the application, based on the dataset of normal observations.

Many algorithms can be used to perform anomaly detection, like the Self-Organizing Map, Statistical, or One-Class Support Vector Machine (OCSVM) [20]. For this first implementation of our prototype, the technique implemented is an OCSVM, using the *scikit-learn* python package. This choice has been driven by the performance of the OCSVM algorithm (mainly its prediction speed) and because it is easy to configure and use.

The OCSVM can be parameterized in different ways:

- $kernel \in \{Linear, RBF, polynomial\}$: Global aspect of the OCSVM.
- $nu \in [0; 1]$: Ratio of acceptable false positives.
- $gamma \in N$: Level of precision around the training points.

E. Attack Injection Tool

The Attack Injection Tool is used to emulate attack consequences on the application through code modification representing malicious behavior. To facilitate and automate the generation of attacks, the debugging interface is used to stop the entire module (RTOS + applications) and modify the code area of the application, to force it to change its behavior after continuing the execution. The tool provides debugging scripts with different attack functions, for example the ability to insert a loop, change the parameters of a function call, modify an API call, or skip or erase a part of the code. The start time and duration of the attack can also be configured. Once prepared, the attack is directly executed from the application.

For example, the *skip_instruction* attack function is defined as follows:

```
# Skip a given instruction
define skip_instruction

# ARG 0 : Address of the instruction to skip
# ARG 1 : N (skip 1 over N instructions)
# ARG 2 : Time limit begin
# ARG 3 : Attack duration


# 1. Jump into the exploit:
# Save skipped instruction and replace it by a
jump
save_instruction $arg0 saved_instruction
put_jmp_exploit $arg0
# Save execution context (registers)
save_registers


# 2. Prepare exploit variables:
# Exploit_size: debugger local variable
set exploit_size = restore_registers_size +
return_size
# Global_counter: start and duration of the
attack
define_global_variable global_counter
# Begin_value and End_value
define_global_variable begin_value $arg2
define_global_variable end_value ($arg3 -
$arg2)
define_global_variable N $arg1


# 3. Start the exploit:
# Increment the global_counter
increment_variable global_counter
# Check if the skip is performed (conditions
on starting, duration, and N)
if_state_jump (begin_value > global_counter)
(if_size*2+exploit_size)
if_state_jump (end_value < global_counter)
(if_size*1+exploit_size)
if_state_jump (global_counter % N)
(exploit_size)
# If no jump is taken, skip the call: restore
the registers and return
restore_registers
return ($arg0+4)
# Else, restore registers, execute the
instruction and return
restore_registers
execute saved_instruction
return ($arg0+4)
```

This function takes as input the instruction to skip, the starting time, the duration, and the skip frequency as parameters. It is executed to prepare the exploit depending on those parameters.

F. SDA Checker

The SDA Checker collects data from the SDA Monitor at runtime. It also imports the SDA built by the SDA Learning Tool. Each data received is pre-processed using the Data Pre-processing Tool to provide a resulting couple [API calls sequence ID, duration]. The one-class SVM classifier identifies this couple as belonging or not to the model exhibited during the training phase. If not, an alarm is raised by the SDA checker. A percentage of anomalies detected by the SDA Checker is also computed by

comparing the log generated by the application execution with the injected attacks, to the log obtained under a normal execution.

G. Conclusions about the prototype

This prototype addresses the different constraints induced by the avionics domain:

- Real-time: The overhead generated by the logging of application events by the RTOS can be controlled in order to reduce the impact on the performance of the RTOS to deliver the API services, and the associated Worst Case Execution Time (WCET) to deliver such service. There are no other impacts on the applications.
- High safety level: If the embedded part of the prototype is implemented in a dedicated partition as suggested, it will be compliant with the spatial and temporal segregation provided by the platform and should not induce dependencies with other applications. Also, at this stage, the HIDS does not have any impact on the flight as the alerts raised are just logged for future investigation. It is not designed to directly interact with the embedded safety mechanisms.
- Performance: The data chosen as input are very simple and are computed very quickly by the OCSVM algorithm. However, the amount of data is not controlled and this should be addressed in a future prototype.
- Maintenance: The SDA needs to be updated at each update of the application, and not at each newly discovered attack. Also, we consider that the knowledge database would require a few updates compared to a classical signature-based IDS.
- Life time: As the SDA is elaborated based on the normal behavior of an application, it should stay accurate during the life time of the application. We consider that the environment of the application is also not going to change.
- Certification: The HIDS itself has no impact on the aircraft safety so a low DAL should be assigned to it. However, the problem of certified algorithms like machine learning ones, has to be addressed to anticipate the actions to take on-board as a reaction to an alert. [29]
- Resilience to attacks: As a system partition developed by the module supplier, this partition should be considered as trusted. Nevertheless, security mechanisms should be considered during the platform development to guarantee the integrity of this partition.

VI.   Preliminary Experiment

For this first experiment, the target is an IMA module with a 32bits PowerPC architecture running a unique application called "IHS" for Interface Human System. This application is responsible for the display in the cockpit of information about the health of the aircraft, like the fuel level or the state of the motors.

The application configuration, the loadable binary, and the insertion contract of this application are available to the module integrator. The application is considered as autonomous for this experiment (it is running dynamically but not stimulated, so that it always displays the same information and has no interaction with the pilots).

This preliminary experiment aims at assessing the relevance of our overall approach in four steps:

- Verify the correct usage of resources of the application (Static Security Analysis)
- Run the application in normal environment and build its SDA (SDA Modeling)
- Inject an attack during the application is running (SDA Validation)
- Investigate the corresponding data offline by comparing it with the application's SDA (SDA Validation)

A. Step 1: Correct Usage of Resources

Table 4 summarizes the information contained in the insertion contract for the IHS application. This information can be verified using the configuration or the code of the application.

Table 4. Resources Defined in the Insertion Contract

Resource Type
RAM size: - Code²   - Data²   - Inter-partition communications¹
Execution Time²
NVM size¹
Number of API communication ports¹
Specific Services¹
Number of Physical Resources : - A429²   - Discrete²
Number of Network Interfaces : - A664²   - Ethernet²

¹ Items checked using the code
² Items checked using the configuration

Elements that can be verified using the application configuration (like the RAM size allocated for code and data or the number of network interfaces), are checked by means of the existing configuration tool. This tool checks whether the configuration provided by the application supplier is consistent with the configuration defined by the module integrator.

The other elements presented in Table 4, are handled by the Resources Usage Analyzer tool using the following formulas as inputs:

- RAM size for inter-partition communications:

```
CREATE_SAMPLING_PORT: MAX_MESSAGE_SIZE
CREATE_QUEUING_PORT: MAX_MESSAGE_SIZE *
MAX_NB_MESSAGE
```

- NVM size:

```
CREATE_LOGBOOK: MAX_MESSAGE_SIZE *
(MAX_NB_LOGGED_MESSAGE +
MAX_NB_IN_PROGRESS_MESSAGE)
CREATE_NOTEPAD: MAX_MESSAGE_SIZE
```

- Number of API ports:

```
CREATE_SAMPLING_PORT: 1
CREATE_QUEUING_PORT: 1
```

- Specific services used: for this part, we look for the API calls corresponding to the list of specific services and display the ones that are called.

To obtain these formulas, the ARINC 653 standard [9] is used. For example, this standard defines two services to perform inter-partition communications: the *SAMPLING* and *QUEUING* ports. Only one message is available in a *SAMPLING_PORT*, and its maximum message size is set as parameter when calling the *CREATE_SAMPLING_PORT* API call. This parameter corresponds to the total size necessary for one declared *SAMPLING_PORT*. For the *QUEUING_PORT*, the standard authorizes multiple messages and its maximum number of messages is given as parameter as well as the maximum message size when using the *CREATE_QUEUING_PORT* API call. The multiplication of these two parameters gives the total size used by a *QUEUING_PORT*.

B. Step 2: SDA Modeling

The target application is first loaded inside the module and the SDA Monitor is parameterized to log the type and timestamp of each called API. The application is run during 25 seconds to have a sufficient amount of data (around 45.000 data lines that cover each part of the code many times). The application initialization is not taken into account in this prototype. The resulting data are pre-processed by the SDA Learning tool to be formatted into [API sequence ID, duration] couples with sequences of length four. This sequence size is chosen arbitrarily for this experiment for the sake of illustration, but other lengths can be considered. The data are also normalized between 0.0 and 1.0 to be easily used by the classifier. These formatted data are then exported into *.csv* files and constitute the inputs of the OCSVM algorithm.
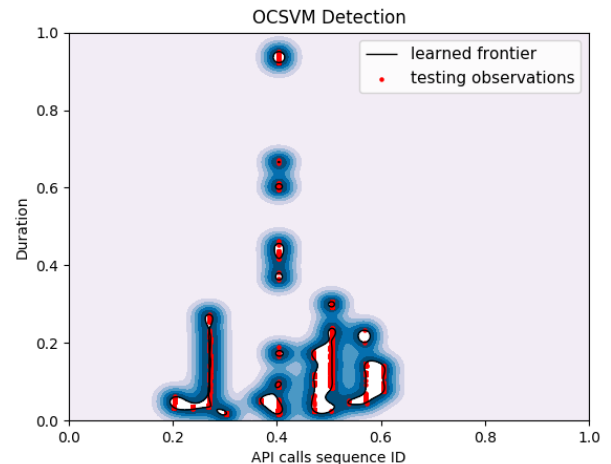

Figure 7. Dataset Obtained by Monitoring the Type and Timestamp of API Call in the Absence of Attack

70% of the dataset is used to train the OCSVM, and 30% are used to test its accuracy. The OCSVM is parameterized with a RBF kernel, $nu = 0.01$, $gamma = 1000$. We obtained 1.45% of false positives on the testing set with this OCSVM.

Figure 7 plots the testing data obtained with the OCSVM trained. This OCSVM is exported in a *.plk* file of size 13,4*Ko*.

C. Step 3: Attack Injection

Using the Attack Injection Tool, we emulated a Denial of Service on the IHS display. This attack has been carried out in two different ways:

- *Full attack:* The IHS display is completely disabled from the starting of the partition,
- *Partial attack:* After one second of normal working, half of the frames are disabled, making the screen blinking during three seconds.

The data collected under injection are collected with the same SDA Monitor script as the one used in the previous step.

D. Step 4: SDA Checking

The attack data are formatted using the Data Pre-processing Tool into `[API calls sequence ID, duration]` couples with sequences of length four, and then exported in a *.csv* file.

The resulting *.csv* file is passed to the OCSVM previously defined and each point of the attack file is evaluated as normal or as an anomaly using the *predict()* function of the *scikit-learn* python package. The results are summarized in Table 5. The OCSVM raised 8.14% of anomalies in the full attack dataset. This rate is significantly higher (ratio of 5.60) than the false positive rates (1.45%) found in the step two on the normal dataset (used for testing). As a consequence, the classifier is able to differentiate the attack from the normal behavior of the application. The anomaly rate is less important in the partial attack dataset (3.29%), but this can be explained by the shorter duration of this attack compared to the full attack.

These results are obtained with only one type of attack injected in two different modes. This preliminary experiment provides some positive and encouraging feedback. In particular, the attack is discriminated whereas it does not induce new API call sequences, meaning that the duration is a very interesting data to monitor in such context. However, more extensive and significant experiments should be carried to assess the detection effectiveness and performance of the proposed HIDS, including a large set of different attacks and a large number of experiments to obtain statistically significant results. The application used for the prototype should also be stimulated to build an SDA covering its overall behavior, including user interactions and fault-recovery situations. The SDA modeling parameters should also be challenged, including nu and gamma parameters, but also the sequence size and the amount of data to use for training. Such experiments are planned for future work.

Table 5. Results: Percentage of anomalies on normal and attacked datasets

Dataset	Anomalies Rate
Testing (Normal)	1.45%
Full Attack	8.14%
Partial Attack	3.29%

VII.   Conclusion and Future Work

This paper discussed the challenges related to the use of traditional IDS techniques in the context of real-time critical avionic systems. We also presented the principles of an anomaly-based intrusion detection approach aimed at modeling and monitoring the behavior of an avionic application through a HIDS process adapted to avionics constraints. The integration of this approach into a conventional IMA process is also described. A prototype of this approach has been developed and some preliminary results have been presented.

In the future, we plan to go further in the development of the prototype. In particular, we are currently implementing other functionalities in the Resources Usage Analyzer to check the consistency between the resources declared and their usage. We are also improving the Attack Injection Tool to formalize different classes of attacks, and create an automatic attack injection campaign, to perform a more extensive validation experimentation. As regards the SDA modeling, we plan 1) to carry out sensitivity experiments in order to evaluate the best parameters for our selected OCSVM algorithm (nu, gamma, kernel, and length of the sequences), and 2) to implement other anomaly detection techniques like statistical ones. We also plan to test other relevant parameters to be monitored by the SDA Monitor. Finally, we plan to improve the prototype by using it on a stimulated IHS application and on other kinds of applications to evaluate the approach on a more realistic environment.

References

1. SAE International, 2010, "ARP4754A: Guidelines for Development of Civil Aircraft and Systems", doi:10.4271/arp4754A.
2. RTCA, 2011, "DO-178C: Software Considerations in Airborne Systems and Equipment Certification".
3. Prisaznuk, P. J., 1992, "Integrated Modular Avionics", In Proceedings of the IEEE 1992 National Aerospace and Electronics Conference@m_NAECON 1992, 39–45 vol.1, doi:10.1109/NAECON.1992.220669.
4. Chen, T. M., and Abu-Nimeh S., 2011, "Lessons from Stuxnet", Computer 44 (4): 91–93, doi:10.1109/MC.2011.115.
5. "Jeep Hacking Incident Leads to Fiat Chrysler Recall of 1.4M Vehicles", 2015, Claims Journal. 27 July 2015, https://www.claimsjournal.com/news/national/2015/07/27/264766.htm.
6. Biesecker, C., 2017, "Boeing 757 Testing Shows Airplanes Vulnerable to Hacking, DHS Says", Avionics, 8 November 2017, https://www.aviationtoday.com/2017/11/08/boeing-757-testing-shows-airplanes-vulnerable-hacking-dhs-says/.
7. Teso, H., 2013, "Aircraft Hacking - Practical Aero Series", presented at the Hack In The Box (HITB) Conference, Amsterdam, April.
8. Conmy, P., Nicholson, M., and McDermid, J., 2003, "Safety Assurance Contracts for Integrated Modular Avionics", 10.
9. Prisaznuk, P. J., 2008, "ARINC 653 Role in Integrated Modular Avionics (IMA)", In 2008 IEEE/AIAA 27th Digital Avionics Systems Conference, 1.E.5-1-1.E.5-10, doi:10.1109/DASC.2008.4702770.
10. Kocher, P., Genkin, D., Gruss, D., Haas, W., et al., 2018, "Spectre Attacks: Exploiting Speculative Execution∗", January, 16.
11. Parkinson, S., Ward, P., Wilson, K., and Miller, J., 2017, "Cyber Threats Facing Autonomous and Connected Vehicles: Future Challenges", March, 18, doi:10.1109/TITS.2017.2665968.

12. RTCA, 2014, "DO-326A_Airworthiness Security Process Specification".
13. RTCA, 2018, "DO-356A_Airworthiness Security Methods and Considerations".
14. Hintze H., and God R., "Using Model-Based Security Engineering in the Development of Complex Aircraft Cabin Systems", *SAE Int. J. Aerosp.* 8(1):2015, doi:10.4271/2015-01-2445.
15. ARINC Industry Activities, 2017, "664P5 Aircraft Data Network, Part 5, Network Domain Characteristics and Interconnection", SAE ITC, October
16. Netkachova, K., Müller, K., Paulitsch, M., and Bloomfield, R., 2015, "Investigation into a Layered Approach to Architecting Security-Informed Safety Cases", In 2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC), 6B4-1-6B4-12, doi:10.1109/DASC.2015.7311447.
17. Dessiatnikoff, A., Nicomette, V., Alata, É., Deswarte, Y., et al., 2013, "Securing Integrated Modular Avionics Computers", In 2013 IEEE/AIAA 32nd Digital Avionics Systems Conference (DASC), 4A3-1-4A3-11, doi:10.1109/DASC.2013.6712577.
18. O'Neill, K., Newell, G. R., and Odiga, S. K., 2016, "Protecting Flight Critical Systems against Security Threats in Commercial Air Transportation", In 2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC), 1–7, doi:10.1109/DASC.2016.7777979.
19. Pharate, A., Bhat, H., Shilimkar, V., and Mhetre, N., 2015, "Classification of Intrusion Detection System", International Journal of Computer Applications 118 (7): 23–26, doi:10.5120/20758-3163.
20. Wu, S. X., and Banzhaf, W., 2010, "The Use of Computational Intelligence in Intrusion Detection Systems: A Review", Applied Soft Computing 10 (1): 1–35, doi:10.1016/j.asoc.2009.06.019.
21. Tabrizi, F. M., and Pattabiraman, K., 2015, "Flexible Intrusion Detection Systems for Memory-Constrained Embedded Systems", In Dependable Computing Conference (EDCC), 2015 Eleventh European, IEEE, doi:10.1109/EDCC.2015.17.
22. Studnia, I., Alata, E., Nicomette, V., Kaâniche, M., et al., 2014, "A Language-Based Intrusion Detection Approach for Automotive Embedded Networks", In The 21st IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2015), Zhangjiajie, China, doi:10.1504/IJES.2018.089430.
23. Yoon, M.-K., Mohan, S., Choi, J., Kim, J.-E., et al., 2013, "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, 21–32. IEEE, doi:10.1109/RTAS.2013.6531076.
24. Kim, G., Lee, S., and Kim, S., 2014, "A Novel Hybrid Intrusion Detection Method Integrating Anomaly Detection with Misuse Detection", Expert Systems with Applications 41 (4): 1690–1700, doi:10.1016/j.eswa.2013.08.066.
25. Om, H., and Kundu, A., 2012, "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, 131–136. IEEE, doi:10.1109/RAIT.2012.6194493.
26. Gil Casals, S., Owezarski, P., and Descargues, G., 2013, "Generic and Autonomous System for Airborne Networks Cyber-Threat Detection", doi:10.1109/DASC.2013.6712578.
27. Jacob, G., Debar, H., and Filiol, E., 2008, "Behavioral Detection of Malware: From a Survey towards an Established Taxonomy", Journal in Computer Virology 4 (3): 251–66, doi:10.1007/s11416-008-0086-0.
28. Bonfante, G., Kaczmarek M., and Marion J-Y, 2009, "Architecture of a Morphological Malware Detector", Journal in Computer Virology 5 (3): 263–70, doi:10.1007/s11416-008-0102-4.
29. Bhattacharyya S., Cofer D., Musliner D., Mueller J., et al., 2015, "Certification Considerations for Adaptive Systems", In 2015 International Conference on Unmanned Aircraft Systems (ICUAS), doi: 10.1109/ICUAS.2015.7152300

Contact Information

Aliénor DAMIEN,
PhD Student in Embedded Security
e-mail: alienor.damien@laas.fr

Abbreviations

API	Application Programming Interface
DAL	Design Assurance Level
HIDS	Host-based Intrusion Detection System
IDS	Intrusion Detection System
IHS	Interface Human System
IMA	Integrated Modular Avionics
NVM	Non-Volatile Memory
OCSVM	One-Class Support Vector Machine
RTOS	Real-Time Operating System
SDA	Security Domain of the Application

7/22/2018