Attack Injection into Avionic Systems through Application Code Mutation

Aliénor Damien, Nathalie Feyt, Vincent Nicomette, Eric Alata, Mohamed Kaâniche

Attack Injection into Avionic Systems through Application Code Mutation

Aliénor Damien*† Nathalie Feyt*, 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*—Given the continuous increase of malicious threats targeting embedded systems, the potential malicious modification of an aircraft application, by exploiting an unknown software or hardware vulnerability of the execution platform, must be seriously considered for future systems. Indeed, an insider attack breaking the organization's security measures to insert a malicious function on board could have significant consequences. Various solutions can be investigated to provide enhanced protection against such threats, including intrusion detection techniques. To design an Intrusion Detection System (IDS), and more specifically to evaluate its performance, abnormal data are required. However, to our knowledge, there is no publicly available attack data for aircraft applications. This paper proposes an approach and a tool aiming at automatically performing application code mutations that mimic the behavior of malevolent pieces of code introduced inside an application. The approach relies on three code modification strategies, designed to cover both generic and specific mutations. The tool takes into account the specific characteristics of avionic applications (dedicated hardware, real-time execution, threat model). This paper describes the architecture and implementation details of the tool, as well as some experiments, in which it is used in order to calibrate a Host-based Intrusion Detection System (HIDS) that we are currently implementing. For that purpose, specific code changes are introduced, targeting application integrity and availability as well as safety.**

I. Introduction

Aircraft systems are constantly evolving, offering more and more connectivity and services to the passengers. Unfortunately, from the security point of view, such evolution leads to a larger attack surface. Furthermore, the threats targeting embedded systems are constantly evolving and the impact of an insider attack breaking the organization's security measures could have significant consequences. In this context, it is essential to anticipate a possible malicious modification of an application for future systems, and to propose security mechanisms to deal with such threats. Among various solutions to provide enhanced protection, intrusion detection techniques are widely used in computer security.

A Host-based Intrusion Detection System (HIDS) relying on anomaly detection techniques has been proposed in previous work [3]. The proposed approach is based on machine learning techniques using semi-supervised algorithms. While such approach only needs normal data to model the correct behavior of an application, abnormal data (i.e., attack traces) are still needed to assess the HIDS accuracy, and to calibrate the parameters of the anomaly detection technique. However, to our knowledge, there is no public attack data on aircraft applications. Such data can be generated experimentally by controlled experiments. This paper therefore proposes the design and implementation of a tool to apply specifically customized modifications to avionic applications code (called fault injections), that mimic the behavior of malevolent pieces of code introduced by potential attackers. This tool is intended to support the validation of HIDS and its integration with avionic applications. Therefore, it should be designed to generate a wide range of malevolent pieces of code and malicious behaviors. The design of the proposed tool takes into account three main assumptions that are inherent to the avionics context considered in our study. We consider the particular case of avionic applications integrators, who, in most cases, do not have access to the source code of the applications. Therefore, fault injection must be carried out in the application binary code. The second assumption is related to the real-time constraints that are essential in avionic applications. Accordingly, the tool must not disturb the real-time execution of the different applications, and must be adapted to the underlying hardware. Finally, the injected faults should be adapted to the threat assumptions that are specific to this context. For example, an attacker may want to execute a malicious code only when the aircraft is in flight.

To provide a large coverage of possible malevolent pieces of code, three different code modification strategies are proposed as part of this approach with the possibility of generating randomly different instances of each strategy. The first strategy allows for a large space of mutations without a precise goal, the second one offers a more restrictive space of mutations without a precise goal, and the third proposes a more restrictive space of mutations with more specific goals. They are intended to cover different types of attack scenarios. While the first strategy is more likely to mimic attack scenarios targeting the availability of an avionics application, the second and third strategies better cover scenarios targeting its confidentiality, integrity, or traceability.

The architecture of a prototype tool implementing this approach is detailed in this paper, as well as some experiments that have been carried out on an avionics application implemented in an hardware-representative environment. Several scenarios have been setup, in order to emulate different types of impacts, such as denial of service, loss of data integrity, and deactivation of safety mechanisms. These experiments allowed us to evaluate the detection performance of an HIDS we are currently developing.

Section II presents some related works dealing with attack and fault injection. Section III describes the proposed approach and presents the architecture of the tool as well as the three attack injection strategies. Section IV provides some implementation details of the tool, while Section V presents the experiments we carried out to evaluate a HIDS with this tool. Finally, Section VI concludes and discusses further improvements.

II. Related Work

In order to assess the efficiency and performance of an IDS, it is necessary to have access to attack data that is actually observed or synthetically generated. Such attack data should cover as much as possible a wide variety of possible attacks. Several frameworks have been defined to characterize attacks targeting traditional computer systems, e.g., [2], or to design experimental campaigns to assess intrusion detection systems, e.g., [7]. However, to our knowledge, only a few studies addressed the specific context of avionics applications and their associated constraints (e.g., [4]).

Generally, it is difficult to collect real attack data, especially from critical embedded systems. Alternatively, several studies focus on the set up of experimental testbeds based on fault injections, which aim to emulate malicious behavior representative of the manifestation of attacks on the target systems. As an example, ID2T [10] injects malicious packets inside network data captured on a real environment to emulate network attacks. Vulnerability and attack injection is also used in [6] to modify the code of a web service (inject the vulnerability), and perform SQL injections (inject the attack) to exploit the vulnerability. To our knowledge, there are no equivalent tools to inject a malicious code into avionics applications.

Fault injection techniques have been widely used to assess fault tolerance mechanisms developed for critical systems, including avionics systems. A survey on Software Fault Injection (SFI) is proposed in [8], presenting in particular the injection of code changes (called mutations) to emulate software faults. Similar mechanisms can be developed when considering attacks. The principles of code mutations correspond to the attack injection we want to implement, with the exception of the following hypothesis:

1) The instructions injected into the binary code do not necessarily have to be representative of faults in the source code
2) Even if the payload introduced by the attack is not activated during the experiment, it must be detected
3) The distribution of faults proposed by [5] is not applicable to intentional faults

[5] proposed a set of 18 operators to cover around 68% of most frequent software faults. These operators have been extended by [1] to cover around 57% of vulnerabilities found in open-source projects. However, these operators are not necessarily representative of malicious pieces of code.

Using existing attack classifications and code injection techniques, this paper aims to 1) propose elementary modification patterns to represent the consequences of an attack on an avionics application, 2) propose an architecture for an injection tool to implement these elementary code changes; this tool has been designed and developed to adapt to specific avionics constraints, such as hardware and real-time constraints and 3) describe the experiments we have conducted, in which this tool is used to calibrate and evaluate a HIDS that we are developing.

III. Approach

The section first presents an overview of our code injection approach (Section III-A), then describes the components we have designed and developed to implement this approach (Section III-B), and finally details the code injection strategies as well as the associated *attack operators* (Section III-C).

A. Overview

The proposed attack injection is similar to fault injection based on code mutation. As mentioned in [8], the fault model for making code changes must define "When to inject", "Where to inject" and "What to inject".

"When to inject" is related to the threat model considered in this study. We assume that the code modification is performed by an attacker before the application is loaded on the avionic platform. It may correspond, for example, to a malicious maintenance operator who modifies the binary before loading it on-board, or to a malicious application developer who adds some code in the application binary before sending it for integration. However, given the context of avionics systems, the attacker may want the payload of his attack to be activated while the aircraft is flying, during the take-off or landing, in the middle of the ocean, or while flying over a specific area. Two alternatives are considered to represent this behavior: 1) Run the malicious payload when the corrupted application is started or 2) Embed an additional piece of code that waits for specific time or event-related conditions to run the payload.

As regards the location of the mutation ("Where to inject"), it must be done within the application code. As the application code area is likely to be very large, it is important to target specific areas where the attack is activated. However, inserting the malicious injected code may require more space than the original memory allocated, and may require the installation of an additional code area. It is therefore necessary to identify a location for this option in the application itself, for example a code area that is unused.

Finally, with regard to the types of mutations injected that should be representative of malevolent pieces of code ("What to inject"), three different strategies for modifying the binary code are proposed. These strategies and the associated attack operators are described in Section III-C.

B. Attack Injection Tool Components

The attack injection campaign consists in defining a list of attacks, executing each of them, and collecting a set of observations to evaluate their impact. The proposed approach, depicted in Figure 1, is based on four main components, and is

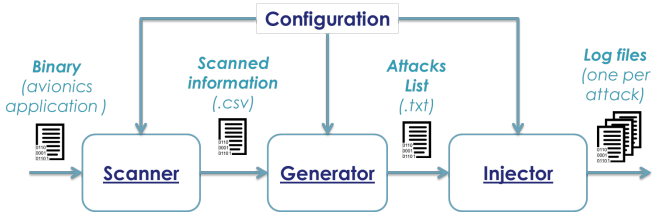

Figure 1. General approach

aimed at randomly generating a list of attacks and performing the corresponding attack injection campaign.

1) Configuration: There are three different configuration levels for the proposed injection tool. The first level defines general information about the targeted application, such as the addresses of code and data memory segments, the binary entry point, or the address of an unused code area. This information can be provided by the Module Integrator or easily retrieved using code analysis tools. The second configuration level is related to the control of the experiments and specifies how to restart or restore the environment between two attack injections, how to load the application on the platform, and also the duration of the experiments. The third level defines the attack operators to be used, the type of parameters for each attack operator, and the information to be logged during the experiments (see Section III-C).

2) Scanner: Scanners are used to establish lists of parameters that are consistent with the different attack operators. We implemented three scanners that respectively scan function entry points, jump instructions, and more generally instructions executed by an application, with a given workload. The result of this step is a set of CSV files containing a list of addresses, corresponding to the scanned information.

3) Generator: This step aims to select a list of attack operators, either randomly or according to the tool's configuration and associated parameters. The value of each parameter is randomly generated according to the type of parameter, using the scanner results if available. For example, an instruction address must be chosen from the list of addresses given by the scanner, while a value must be a 32-bits random number. The results of this step is a text file containing a list of code changes, that consist of a list of attack operators with their associated parameters.

4) Injector: The role of the injector is to automatically launch the attack injection campaign, using a list of attacks, and to monitor the application during each experiment. To manage the attack injection campaign, for each attack until the entire list is executed, the injector 1) launches the current attack, 2) checks the end of the experiment (according to the duration configured previously), and 3) restores the initial state of the module and restarts it. For each experiment, a log file is generated at the end of the application execution according to the previously configured observation points.

Table I
ATTACK OPERATORS

Action	Comment
Randomly Modify Instruction (RMI)	Replace current instruction by a random value
Modify Instruction (MI)	Replace current instruction by another
Modify Branch (MB)	Replace current branch instruction by another
Modify Call (MC)	Replace current function call by another
Modify Register (MR)	Replace register value by another
Modify Value (MV)	Replace stored data value by another
Add Instruction (AI)	Insert en instruction
Add Branch (AB)	Insert a branch instruction
Add Call (AC)	Insert a function call instruction
Remove Instruction (RI)	Replace the instruction by a NOP
Remove Branch (RB)	Replace the branch instruction by a NOP
Remove Call (RC)	Replace the function call instruction by a NOP

C. Strategies of Code Injection and Associated Attack Operators

As no attack database is currently available to our knowledge, we need to create artificial examples of malevolent pieces of code. This malevolent code can be either completely different from the classical avionic functions, or just a slight modification that executes correctly. Three different strategies detailed below, namely "CrashMe", "Well-formed Instruction Substitution" and "Attack Pattern", are adopted to perform changes in the binary code. They have been designed either to cover a large amount of possible modifications, leading to possible incorrect executions, or to generate well-formed malicious code that can be correctly executed.

The 12 attack operators described in Table I implement these strategies. In particular, the "Randomly Modify Instruction" operator is associated with the "CrashMe" strategy, the "Modify Instruction" operator is associated with the "Well-formed Instruction Substitution" strategy, and the remaining operators are associated with the "Attack Pattern" strategy.

1) Crash Me: The most generic approach consists in replacing some instructions with a random code. This is the principle of the "CrashMe" strategy. Such an approach has already proven effective in carrying out Denial of service (DOS) attacks or in identifying some weaknesses in safety detection mechanisms, as presented in [4]. This code injection strategy is implemented through a single attack operator that consists in replacing an instruction with a random code (RMI in Table I). Such changes are easy to make, anywhere in the binary. Moreover, this technique covers a large number of scenarios that potentially use incorrect instructions. However, it cannot be used to implement meaningful attack scenarios like data exfiltration or data falsification. The code mutations generated are mostly intended to cause a crash of the application, i.e., a DOS attack. As a result, our tool also implements two other code modifications strategies, aiming at making more significant mutations.

2) Well-formed Instruction Substitution: The second technique implemented by our tool is based on a pre-constructed dictionary of well-formed instructions, selected from the application itself, other applications, and/or kernel code. This strategy replaces the application's code by well-formed pieces of code, i.e., code parts composed of valid binary instructions but not intended to mimic a specific behavior of an attack. Even if these code changes are not developed to emulate some real attacks, they allow to generate a wide variety of application mutants with a different behavior than the original application. In addition, since code changes are made with valid instructions, the corresponding mutants are less likely to cause a crash than in the "CrashMe" approach. This code injection strategy is implemented through a single attack operator, which consists in replacing one instruction with another (MI in Table I).

3) Attack Patterns: To further define malicious behavior, we implemented different attack scenarios and extracted elementary operations performed by the corresponding malevolent codes. These attack scenarios included communication tampering (payload, header, configuration), process tampering (period, code executed), information leakage (configuration location, communication ports), and were performed on a test application different from the target application used in Section IV. Then, we defined attack operators associated with these elementary operations (MB, MC, MR, MV, AI, AB, AC, RI, RB, RC in Table I). They include basic operations (Modify, Add, or Remove) performed on a target (Instruction, Branch, Call, Register, or Value). Such modifications allow the generation of mutants whose code includes payloads that are actually representative of the payload of real attacks. Concerning the implementation of these operators, the user must define the type of parameters to use. The tool is then able to select a random value in an appropriate range, according to the type of parameter and scanner results (see Section III-B).

IV. Prototype

This section details the prototype we implemented, based on the architecture presented in Section III. Figure 2 presents the components of the prototype as well as their interactions. The target is an avionics system under development, running on a PowerPC architecture with a T2080 processor. The target application is a cockpit display application called HSIV for Human-System Interface Vehicle. This application displays information to the pilot about the state of the aircraft, such as fuel level or tire pressure. Only one workload is used, which always sends the same data to be displayed by the application. The workload is implemented in the application itself. A computer (called Controller in Figure 2) is used to communicate with the target through Ethernet, using a GNU Debugger[1] (GDB) Client-Server communication. An embedded GDB server runs on the target. This embedded GDB server is directly linked to the kernel, so that a breakpoint stops the execution of the target. The GDB server is specifically

[1]https://www.gnu.org/software/gdb/

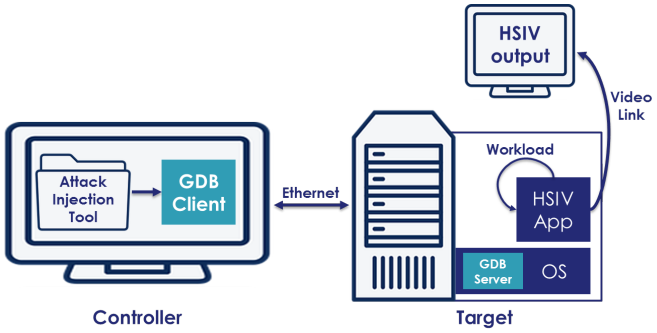

Figure 2. Architecture of the Prototype

designed to preserve the value of two internal registers *$tbl* and *$tbu* indicating the clock value of the target, so that the time between two breakpoints is guaranteed with an approximation of 5 usec.

The Attack Injection Tool is implemented as a set of GDB and Python scripts. GDB scripts are used to interact with the target (start the application, extract logs, inject attack, restart the module, ...), while python scripts are used to start the scanners, generate the list of attacks, and generate experiment-specific GDB scripts. Code mutations are performed during the initialization phase of the module. A breakpoint is set at the initialization, and when the breakpoint is reached, the modification is made directly in the RAM area using GDB.

A. Addition of Code

To insert an additional piece of code into the application, it is necessary to identify a free code area to install the additional code. This area is defined by the application-specific configuration file. The instruction to be modified is replaced by a jump instruction pointing to this free zone. At the end of the payload code, another jump instruction is added to return to the instruction following the modified instruction.

B. Control of the Payload Execution

The piece of code added to control the payload execution corresponds to the following pseudo-code:

```
counter++;
# Check the counter value
if first < counter < last:
  # Check the frequency
  if counter % N == 0:
    payload
```

A counter is defined in a free data area and incremented each time the attack is activated. Three parameters are defined for the attack operators, corresponding respectively to the counter value that triggers the payload execution for the first time, the counter value that triggers the end of the payload execution, and the frequency of the payload execution (the payload is executed once over N attack activations).

Table II
CHARACTERISTICS OF THE IMPLEMENTED ATTACK OPERATORS

Operator	Strategy	Code Addition	Execution Control	Frequency
RMI	1			
MI	2			
MB_1/MB_2/RI	3			
AI	3	X		
MR/MV	3	X	X	
RC	3	X	X	X

C. Attack Operators Implemented

In order to implement different schemes, the operators are designed to execute their payload at anytime or under time conditions, with or without addition of code, and with the three different code modification strategies proposed. Table II summarizes the implemented attack operators and their corresponding characteristics. Four characteristics are distinguished. "Strategy" indicates the previously defined strategies (1 = CrashMe, 2 = Well-formed Instruction Substitution, 3 = Attack Patterns). "Code Addition" indicates whether or not code has been added for this operator. "Execution Control" indicates whether or not the time of activation of the exploit is controlled. "Frequency" indicates whether or not the user can select the frequency of the payload execution.

D. Attack Injection Campaign

A python script is used to generate the GDB script to perform the entire attack injection campaign. The following code is an example of such generated GDB script, and is composed of different parts details further.

```
# 1. Observation Breakpoints Definition
# HSIV_ENTRY_POINT
b *0x30000000
# GET_TIME
b *0x10000000
# SEND_QUEUING_MESSAGE
b *0x20000000

# 2. Initializations
# LOCAL VARIABLES FOR ATTACK CAMPAIGN
set $MAX_COUNTER = 4
set $curr_counter = -1
# CONTINUE TO REACH A FIRST BREAKPOINT
continue
# Loop to launch ATTACK automatically
while ($curr_counter < $MAX_COUNTER)

  # 3. Catch breakpoints
  if $pc == 0x10000000 || $pc == 0x20000000 || $pc
  ↪  == 0x30000000 || $pc == $addr_attack
    # Restore and restart the module
    if  $tbl >= $max_duration_tbl
      set logging off
      clear *$addr_attack
      restore_and_restart_module
    end
    # HSIV_ENTRY_POINT
    if $pc == 0x30000000
      retrieve_and_inject_attack
    end
    # GET_TIME
    if $pc == 0x10000000
      printf "1,0x%x,0x%x\n",$tbu,$tbl
    end
    # SEND_QUEUING_MESSAGE
    if $pc == 0x20000000
      printf "2,0x%x,0x%x\n",$tbu,$tbl
    end
    # ATTACK ADDRESS REACHED
    if $pc == $addr_attack
      printf "ATTACK_LAUNCHED,0x%x,0x%x\n",$tbu,$tbl
    end
  # Crash of the application
  else
    # Restore and restart the module
    set logging off
    clear *$addr_attack
    restore_and_restart_module
  end
  continue
end
```

1) Breakpoints declaration: The script starts by declaring a few breakpoints according to the configuration. In the configuration file presented below, three breakpoints are configured:

```
0x30000000;PLACE_ATTACK;HSIV_ENTRY_POINT
0x10000000;1;GET_TIME
0x20000000;2;SEND_QUEUING_MESSAGE
```

The first line indicates that the current attack is injected when the application's entry point is reached, corresponding to the address *0x30000000*. The second and third lines correspond to observation breakpoints at the addresses *0x10000000* and *0x20000000*, (API calls addresses *GET_TIME* and *SEND_QUEUING_MESSAGE*, with the respective IDs *1* and *2*).

2) Execute in a loop until the end of the campaign: The script sets the number of attacks declared in the attack list, and executes a loop until all attacks are completed.

3) Breakpoints management: When a breakpoint is reached, the script first checks the duration of the current experiment. If it is longer than the duration planned in the configuration, the current attack location breakpoint is removed, and the module is restored and restarted. Otherwise, the script checks if the current execution address (stored in the *$pc* register) corresponds to an intended breakpoint address. If the address matches the *PLACE_ATTACK* tag, the script injects the current attack. If the address corresponds to an observation point, the script adds a log to the current log file, along with the configured ID and a timestamp (stored in the *$tbu* and *$tbl* registers). If the address corresponds to the location of the attack, a log is also added. If the address is unknown, the execution should have encountered an exception (crash of the application). The script stops the current attack and continues the campaign with the next one.

4) Attack Injection: To inject the current attack, a GDB script is generated depending on the current attack counter value. The following code is an example of such GDB script:

```
set $curr_counter = $curr_counter + 1
set $addr_attack = 0x40000000
b* $addr_attack
```

```
AI 0x40000000
set logging file data_AI-0x40000000.txt
set logging on
```

The first line increments the current attack counter. The next two lines configure the current location of the attack and place a breakpoint at this location. The fourth line injects the attack payload inside the HSIV application. In this example, the *AI* operator (see Table II) is used to duplicate the instruction at the address 0x40000000. The last lines configure the new logging file.

5) Resulting Log Files: The log file generated for an attack contains a list of logs, alternatively with the ID 1, 2, or the *ATTACK_LAUNCHED* tag. The following example is an extract of a log file.

```
2,0x0,0x14b444d2
1,0x0,0x14b449da
1,0x0,0x14b44fe6
2,0x0,0x14b4535d
ATTACK_LAUNCHED,0x0,0x14b45563
1,0x0,0x14b46e06
2,0x0,0x14b470cb
2,0x0,0x14b47441
```

V. Experimental Results : HIDS Evaluation

The tool was used to assess the efficiency of an HIDS that we are currently developing. The following sections briefly present the HIDS (Section V-A), and how the injection tool is used 1) to calibrate the parameters of this HIDS through experiments based on generic attacks (Section V-B), and 2) to evaluate the performance of the calibrated HIDS when specific targeted attacks are injected (Section V-C).

A. HIDS principle

Although the detailed description of the HIDS that we are currently designing is out of the scope of this paper (see [3] for more details), we briefly introduce the principle of the underlying detection approach. This HIDS is based on anomaly detection and aims to build a model of the normal behavior of an avionics application by observing and characterizing the different syscalls that this application executes (syscall identification as well as the duration between some syscall sequences are considered). Figure 3 outlines the main components of the HIDS with the corresponding parameters. The log files generated by the attack injection tool are used as input for this HIDS. The first step, *preprocessing*, consists in building the sequences of syscalls with the corresponding duration from a log file containing a list of syscall IDs performed by the application and associated timestamp. One normal preprocessed file describing the application's behavior in the absence of attacks is used to *train* a One-Class Support Vector Machine learning model. The resulting model is used to *test* the remaining preprocessed files to determine, for each file, whether an attack, corresponding to a significant deviation of the observed syscalls sequences or their duration from the normal learned preprocessed file, is detected.

Table III
Composition of the Randomly Generated Attack List

Operator	Number	Number with *ATTACK* tag
MI	20	16
MB_1	6	6
MB_2	3	0
MR	15	14
MV	3	3
AI	16	12
RI	11	7
RC	10	3
Total	84	61

B. HIDS Parameters Evaluation

The attack injection tool was used to generate an attack injection campaign consisting of 84 attacks. These attacks are referred to as "random attacks" in the following, because the operators and associated parameters were randomly selected from the list of attack operators implemented (except the RMI operator). Among the 84 log files generated, only 61 contained the tag *ATTACK_LAUNCHED*, meaning that the attack is activated. This is due to the first implementation of our instruction scanner. Indeed, the scanner currently implemented lists the functions that are executed and considers any instruction of each function as a potential target for code injection. Depending on the workload, some part of the code may never be executed (for example, branch never taken). This leads to an attack injected but never executed because the injection point is never reached with this workload.

Only these 61 log files were considered for the results presented in the following. The attack operators randomly selected for the attack campaign are given in Table III. As this is a randomly generated campaign, the values vary from an experiment to another. The duration of each attack is 20 seconds.

Also, 5 log files of normal behavior have been captured, one of 20 seconds, three of 30 seconds, and one of 40 seconds. The real duration to capture 20 seconds of normal behavior was about 60 minutes. A set of 22 observation points has been monitored, corresponding to the 22 ARINC 653 [9] API calls used by the application. All these log files have been used as input data for the previously described HIDS. The parameters have been customized to achieve an F-measure of 97.03% on the testing files. More precisely, the 5 normal files were detected as normal, 49 attack files were detected as containing an attack, and the HIDS did not raise any alarm for the 3 remaining attack files (they were considered as normal).

C. HIDS on Targeted Attacks

To assess the accuracy of the HIDS against realistic attacks, we have developed a set of mutant applications that have a real impact on the display. These attacks, detailed in Table IV, are called "targeted attacks" in comparison to the previous "random attacks" used to calibrate the HIDS. The targeted attacks have an impact on the safety, integrity, or availability of the application.

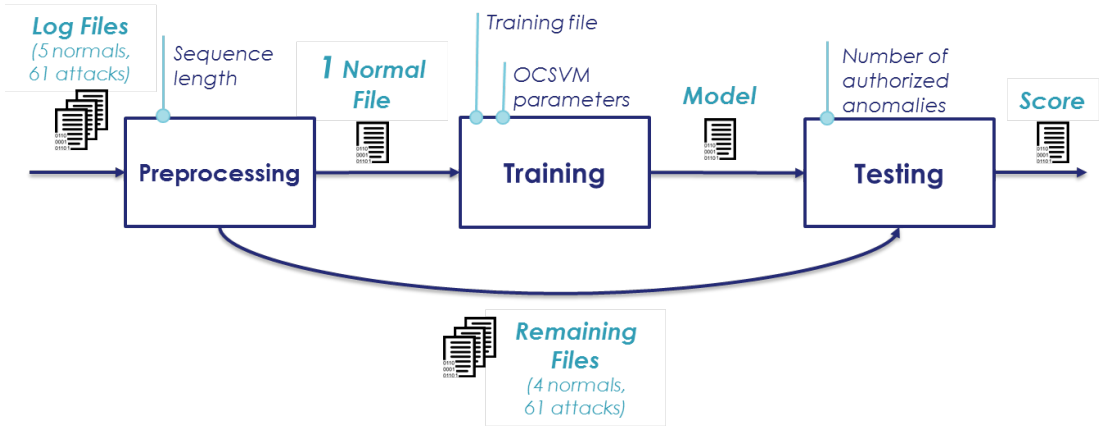

Figure 3. Process to Evaluate the HIDS Parameters

Table IV
TARGETED ATTACK SCENARIOS ON HSIV DISPLAY

ID	Impact	Description
1	Safety	Suppression of the calls to "RAISE_APPLICATION_ERROR" service
2	Integrity	Modification of the displayed value of a gauge
3	Integrity	Constant modification of the names of menus displayed
4	Integrity	Partial modification of the names of menus displayed
5	Availability	Suppression of the calls to "UNLOCK_PREEMPTION" service
6	Availability	Partial suppression of the calls to "UNLOCK_PREEMPTION" service
7	Availability	Constant blinking of the screen
8	Availability	Partial blinking of the screen (from 2 seconds and for 3 seconds)
9	Availability	Destruction of the structure of the display

Each attack was executed 5 to 7 times, resulting in a global set of 51 log files. These files are pre-processed and evaluated by the HIDS, resulting in a detection rate of 100% (e.g. each log file was detected as containing an attack). This result shows that the calibration of our HIDS with the designed attack injection tool and campaign was efficient for detecting real-world examples of attacks. This is very important in an avionics context where, to our knowledge, no attack data is publicly available. However, this result should be confirmed by running a larger number and a wider variety of emulated attacks (each attack should be easily implemented using one attack operator).

VI. CONCLUSION AND FUTURE WORK

This paper presented a tool to inject attacks adapted to an avionics application and an avionics context. In particular, we have proposed a set of attack operators to create randomly generated mutants that represent malevolent applications. The architecture and implementation of the tool have been detailed using an example where the tool is used to calibrate the parameters of an anomaly-based HIDS without known examples of attacks. The experiments carried out with generated mutants, that have been designed specifically for the avionic application under test, showed that the HIDS was able to detect all the targeted attacks.

However, in the current implementation of our prototype the generation of log files may take a long time due to the many breakpoints used to log information. One possible extension should be to instrument the code directly, which would lead to a drastic reduction in the overhead introduced by the breakpoints. Some target avionic platforms already have instrumentation facilities that could be interfaced directly with the tool.

It would also be interesting to test the reaction of safety mechanisms according to the different attack operators proposed. This would help the user to focus on attack classes that are not already covered by safety mechanisms.

Finally, the attack list is currently randomly generated because, to our knowledge, there is no publicly available attack data for aircraft applications. The usage of coverage-guided fuzzing targeting specific behavior of application functions could be an interesting direction to explore to build a more comprehensive and relevant attack list. This would limit the number of cases to test while improving the coverage of the list of attacks.

REFERENCES

[1] Frederico Cerveira, Raul Barbosa, Marta Mercier, and Henrique Madeira. On the Emulation of Vulnerabilities through Software Fault Injection. In *2017 13th European Dependable Computing Conference (EDCC)*, pages 73–78, Geneva, September 2017. IEEE.

[2] The MITRE Corporation. Mitre att&ck™. https://attack.mitre.org/, 2018. Accessed: 03-Apr-2019.

[3] 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.

[4] A. Dessiatnikoff, Y. Deswarte, É Alata, and V. Nicomette. Potential Attacks on Onboard Aerospace Systems. *IEEE Security Privacy*, 10(4):71–74, July 2012.

[5] Joao A. Duraes and Henrique S. Madeira. Emulation of Software Faults: A Field Data Study and a Practical Approach. *IEEE Transactions on Software Engineering*, 32(11):849–867, November 2006.

[6] J. Fonseca, M. Vieira, and H. Madeira. Vulnerability #x00026; attack injection for web applications. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, pages 93–102, June 2009.

[7] A. A. El Kalam M. S. Gadelrab and Y. Deswarte. Defining categories to select representative attack test-cases. In *Proceedings of the 2007 ACM workshop on Quality of protection - QoP'07, Alexandria, Virginia, USA*, 2007.

[8] Roberto Natella, Domenico Cotroneo, and Henrique S. Madeira. Assessing Dependability with Software Fault Injection: A Survey. *ACM Computing Surveys*, 48(3):1–55, February 2016.

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

[10] Emmanouil Vasilomanolakis, Carlos Garcia Cordero, Nikolay Milanov, and Max Muhlhauser. Towards the creation of synthetic, yet realistic, intrusion detection datasets. In *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, pages 1209–1214, Istanbul, Turkey, April 2016. IEEE.