1-1-2016

Model Based Security Testing for Autonomous Vehicles

Seana Lisa Hagerman
University of Denver

Model Based Security Testing for Autonomous Vehicles

Abstract

The purpose of this dissertation is to introduce a novel approach to generate a security test suite to mitigate malicious attacks on an autonomous system. Our method uses model based testing (MBT) methods to model system behavior, attacks and mitigations as independent threads in an execution stream. The threads intersect at a rendezvous or attack point. We build a security test suite from a behavioral model, an attack type and a mitigation model using communicating extended finite state machine (CEFSM) models. We also define an applicability matrix to determine which attacks are possible with which states. Our method then builds a comprehensive test suite using edge-node coverage that allows for systematic testing of an autonomous vehicle.

Document Type

Dissertation

Degree Name

Ph.D.

Department

Computer Science and Engineering

First Advisor

Anneliese A. Andrews, Ph.D.

Second Advisor

Nathan Sturtevant

Third Advisor

Wenzhong Gao

Keywords

Autonomous vehicles, CEFSM, Communicating extended finite state machine, Mitigation patterns, Model based test, Security testing

Subject Categories

Computer Sciences

Model Based Security Testing For Autonomous Vehicles

A Dissertation

Presented to

The Faculty of the Daniel Felix Ritchie School of

Engineering and Computer Science

University of Denver

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

---

by

Seana L. Hagerman

November 2016

Advisor: Anneliese Andrews

Author: Seana L. Hagerman
Title: Model Based Security Testing for Autonomous Systems
Advisor: Anneliese Andrews
Degree Date: November 2016

Abstract

The purpose of this dissertation is to introduce a novel approach to generate a security test suite to mitigate malicious attacks on an autonomous system. Our method uses model based testing (MBT) methods to model system behavior, attacks and mitigations as independent threads in an execution stream. The threads intersect at a rendezvous or attack point. We build a security test suite from a behavioral model, an attack type and a mitigation model using communicating extended finite state machine (CEFSM) models. We also define an applicability matrix to determine which attacks are possible with which states. Our method then builds a comprehensive test suite using edge-node coverage that allows for systematic testing of an autonomous vehicle.

Acknowledgments

I would like to express my sincere gratitude to Dr. Anneliese Andrews. As my advisor, she spent countless hours guiding, directing, mentoring and motivating me throughout my research and in preparation for this dissertation. I will always admire her devotion in my pursuit to complete my Ph.D.

I would like to thank my examining committee members Dr. Nathan Sturtevant, Dr. David Wenzhong Gao and Dr. Andrew Linshaw for accepting my request to serve on my oral defense committee. I would also like to thank Dr. Matthew Rutherford for his support in my oral qualifying and preliminary exams.

Special thanks to my friends and student colleagues Dr. Ahmed Gario, Dr. Salwa Elakeili, Steven Oakes and Dr. Julia Varnell-Sarjeant for being on this journey with me and always encouraging me. I would also like to acknowledge the National Science Foundation ROSE HUB Research Center for supporting my research and this dissertation.

I would like to thank Lockheed Martin Space Systems Company for supporting my graduate and research work by funding both my Masters and Ph.D. classes and research credits. My deepest gratitude goes to Dr. Steven C. Smith who believes in me and provides me with endless opportunities in my professional career. I appreciate my former colleagues Betty Glass and Marcia Edwards for teaching me how to be a kind and thoughtful manager.

My thoughts and deep gratitude go to my husband, Ty Hagerman and our children Ashley, Tyler, Hunter, Hope and Riley for their love, encouragement and support throughout this journey. I also appreciate them being patient as I have spent numerous hours working on this dissertation and traveling to attend conferences and present papers.

Table of Contents

List of Figures

List of Tables

Chapter 1: Problem Statement

1.1 Introduction

As autonomous systems are becoming more popular in both military and commercial use, malicious attacks that threaten them are also getting more sophisticated. Autonomous systems generally have some remote piloted capability or rely on navigation systems such as a global positioning systems (GPS) which makes them vulnerable to cyber security attacks. This dissertation demonstrates an approach for security testing autonomous systems. Our approach has been demonstrated on both pre-flight and post-liftoff launch vehicles, as well as an unmanned aerial vehicle (UAV) system. This approach proposes a black box testing method that uses a behavioral model, an attack type and a mitigation model to build a security test suite. We identify attack points in a system using behavioral and attack models. We then generate a UAV security test suite based on attack mitigation models.

Our approach focuses on finding vulnerabilities at the external interface or where internal components interact. Previous research has found that even though individual software components have undergone extensive testing, their interactions with other software components still need to be extensively tested [74]. Autonomous vehicles are also comprised of embedded systems with inherent vulnerabilities. Interactions between these components give rise to security concerns. Security test cases can then be derived from attack scenarios based on complex attack path and coverage

criteria [42]. Test cases are then applied to test the security of the system. Based on the results of these test cases, the system can be modified to increase the level of security.

There are many examples of autonomous systems being compromised by security attacks. On June 4, 1996, the maiden voyage of the Ariane 5 launch vehicle exploded during liftoff. The European Space Agency (ESA) had spent ten years and seven billion dollars developing the launch vehicle. The explosion caused an estimated loss of 500 million dollars and Europe's opportunity to become a front runner in commercial space ventures. The explosion was caused by a software error when a 64 bit variable was written into a 16 bit memory location. This overflow caused complete loss of guidance and altitude information on the Ariane 5 launch vehicle[1]. While this error was not a malicious attack, attackers are capable of injecting data into a system and causing it to malfunction. This is a very clear example of the importance of security testing autonomous systems.

In April of 2005, hackers penetrated the secure network of National Aeronautics and Space Administration's (NASA's) Kennedy Space Center. Space Shuttle Discovery's launch data had been gathered by a program known as *stame.exe*, from the Vehicle Assembly Building (VAB). The attackers transported sensitive pre-launch data to a computer system in Taiwan. By December of 2005, *stame.exe* had spread to a NASA satellite control complex in Maryland and mission control at Johnson Space Center (JSC). The attackers had retrieved 20 gigabytes of compressed data from JSC alone. It took over seven months for authorities to discover the data breach. The stolen data contained design information for the rocket engine and the fuel systems [33].

In December of 2011, the Iranian government claimed to have captured an American RQ-170 Sentinel unmanned aerial vehicle (UAV). An Iranian engineer alleged the

attack occurred when they jammed the land-oriented and satellite control signals, then spoofed the UAV with false global positioning system (GPS) data and forced it to land. The Iranians have further claimed to have decrypted data from the UAV including maintenance and intelligence information [69]. The United States government denied the Iranian claims, but admitted that a UAV was lost in Iran. Regardless of how it was captured, there are documented vulnerabilities in GPS. Therefore, security testing is critical for UAV's.

In November 2012, Japan Aerospace Exploration Agency (JAXA) reported that a computer virus had resulted in a leak of rocket data from its M-V, H-IIA and H-IIB rockets. JAXA suspected the attackers were interested in the solid fuel rocket data, which could be applied to an intercontinental ballistic missile (ICBM) [2][4][23].

In these cases, hackers were suspected to be foreign government agencies that were seeking to use the stolen data (or vehicle) for espionage purposes. To prevent security attacks on launch systems and autonomous vehicles, a systematic method of testing security attacks and mitigations is essential.

1.2 Software Security Test

Software has inherent vulnerabilities that must be addressed at all phases of its life cycle. Vulnerabilities in software are often caused by lack of poor coding standards [14]. The goal of software security is to protect data from threats that could intentionally or inadvertently alter the flow of data. Attacks on software can cause loss of integrity, financial loss, or even loss of life.

To secure software, developers must perform risk management to evaluate situations to ensure that little or no damage is being done to the system's confidentiality, integrity and availability [102]. There are various types of threats to software as de-

tailed in Appendix A as well as software threat models detailed in Appendix B.1. While traditional software testing focuses on common cases, security testing must focus on uncommon cases in which there are very few test cases [14]. This dissertation will focus on malicious software security attacks or cyber security attacks including:

- Denial of Service (DOS) Radio Frequency (RF) Jam Attack - RF Signal ranges are in the public domain, an attacker can figure out which range of RF signal an autonomous system is transmitting on and send signals in the same range overloading an autonomous vehicle with RF Signals and confusing the system creating a condition called a DOS attack, in which the system is confused and does not know which signals to respond to.

- DOS Flood Attack - An attacker floods an autonomous system with commands or data confusing the system.

- Sniff and Spoof Attack - An attacker listens to the data coming from an autonomous system long enough to understand the packet data (for example an attacker might notice that navigation packets are short and video streaming is a larger data packet). Once the packets are understood, the attacker can attempt to Spoof packets similar to the ones it gathered from the vehicle causing an autonomous system to change course or become confused.

- Man in the Middle (MITM) Attack - The attacker gathers data from an autonomous system and passes on either real or modified data to its intended recipient without being recognized.

- GPS JAM Attack - While widely used, GPS has known vulnerabilities in which positioning data can be maliciously altered to cause a vehicle to become confused or crash.

1.3 Model Based Test

Model-based testing (MBT) is a state of the art method for generating test cases on black box systems [26][78][79]. Models can be generated on a system without having all of the intricate design detail used to design a complicated system [93]. Testing all possible paths in a complex software system can be a very large task, depending on the complexity of the system [93]. MBT models are designed to test the functional or intended behavior. Utting et al.[92] classify MBT notations as State-Based, History-Based, Functional, Operational, Stochastic, and Transition-based. Transition-based notations are graphical node-and-arc notations that focus on defining the transitions between states of the system such as variants of finite state machines (FSMs), extended finite state machines (EFSMs), and communicating extended finite state machines (CEFSMs)). More examples of transition-based notations also include Unified Modeling Language (UML) behavioral models (like activity diagrams, sequence and interaction diagrams), UML state charts, as well as Simulink Stateflow charts [93]. Each type of model has its own characteristics. For example, traditional FSMs do not contain a communication mechanism and do not precisely model the interactions of systems [56]. CEFSMs [21] extended finite state machines (EFSMs) to include variables, operations based on variables, and interactions of variables. MBT is designed to test the intended behavior of software. To use MBT for security testing, we will model the behavior, threat and mitigation as independent models and derive test paths that will be woven together. Common modeling tools include Finite state machines (FSMs) [9], Extended Finite State Machines (EFSMs), CEFSMs [59], Petri Nets [101] and UML models [95]. FSMs have been used for over 40 years to mathematically model the behavior of a system [9]. Coverage criteria such as edge, node, edge-pair, prime path, and W-method were shown using an FSM model [7][37][78].

MBT has also been used to characterize black box behavior to build a test suite [92]. The three main steps in MBT are creating a functional test model, identifing test generation criteria and building the test suite [78].

MBT has been used to automatically generate test cases for security testing [79]. Schieferdecker *et al.* [79] cover the types of models in model based security testing (MBST) including architectural and functional models, threat, fault, risk models, and weakness and vulnerabilities models. They also discuss activities involved in MBST which include identifying security test objectives and methods, designing functional test models, determining test generation criteria, generating the tests, and assessing the test results. The paper [79] also addresses an approach in the DIAMONDS project which focuses on risk-based MBST and model-based fuzz testing. However, they do not present systematic testing of attack mitigation.

1.4 Research Questions

This dissertation is focused on a model based test approach for generating a security test suite for autonomous systems. It will address the following research questions:

- RQ1: Can we leverage an MBT behavioral test suite to build a security test suite?
- RQ2: Can we model required mitigations for security attacks and generate mitigation tests?
- RQ3: Can we identify criteria for covering attack scenarios in a systematic way?
- RQ4: Can we use behavioral tests, mitigation test and attack scenarios to build a systematic approach to security testing (MBST)?
- RQ5: Will the approach be scalable?

- RQ6: Will the approach work for single or multiple attacks?

- RQ7: Is the approach robust enough to be applied to various types of autonomous vehicles and launch systems?

The remainder of this document is organized as follows. Chapter 2 describes related work. Chapter 3 offers an approach to generate a security test. Chapter 4 describes our approach applied to case studies. Chapter 5 offers some suggestions about future work. Finally Chapter 6 gives the conclusions.

Chapter 2: Background

This chapter provides previous research in the areas of Model Based Testing, Security Testing and Mitigation Patterns. We will address publications on each of the topics individually and demonstrate that our approach is novel and covers all three areas of the topics. Our research will show that the other publications do not answer our research questions stated above. Our approach fits into the black region illustrated in Figure 2.1.

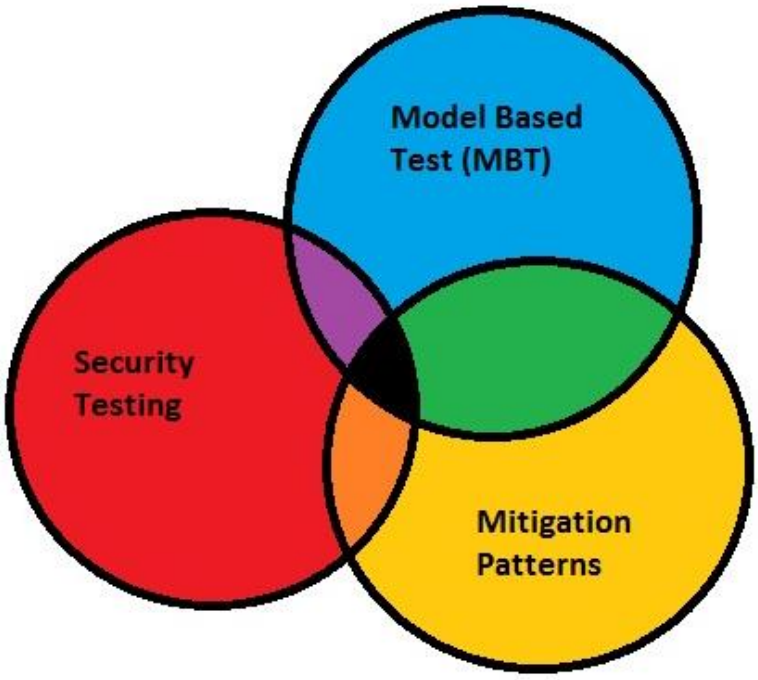

Figure 2.1: Background Venn Diagram

2.1 Background Research

The goal of this background research is to identify key papers as well as identify papers that intersect more than one key topic as shown in Figure 2.1. There were no papers found that intersect all three key areas.

2.2 Model Based Testing (MBT)

Model-based testing uses functional requirements to generate behavioral models which are used for test. Common modeling tools include Finite state machines (FSMs) [9], Extended Finite State Machines (EFSMs), CEFSMs [59], Petri Nets [101] and UML models [95]. FSMs have been used for over 40 years to mathematically model the behavior of a system [9]. Coverage criteria such as edge, node, edge-pair, prime path, and W-method were shown using an FSM model [7, 37, 78]. MBT has also been used to characterize black box behavior to build a test suite [92]. Schieferdecker *et al.* [78] provide a history of model-based testing.

Table 2.1: MBT Research Matrix

Model	Research Area	
	Functional Test	Security Test
UML	[3] [45] [40] [72] [70] [39] [71] [30] [85] [65] [89] [52] [24] [16]	[46] [47] [68] [95]
FSM	[76] [90] [60] [36]	
EFSM	[28] [84] [48]	[81]
CEFSM	[59] [21] [19] [32] [18] [43] [44] [54]	

2.2.1 Unified Modeling Language (UML)

UML is a standard tool for visualizing the design of a system including specifying, analyzing, modeling or documenting software solutions [67, 75]. While UML is generally used for software design it has also been used to model hardware systems. It was developed with the intention of using a graphical approach to more easily visualize the design and interactions of a software system. UML can be used to design a system including identifying components of the system, how the system will run, how the components interact and external interfaces. Examples of UML in design are detailed in [15, 55]. We will focus on UML test [40, 72, 70, 39, 71, 30]. The UML language has simplified the process of converting specifications to test cases and models [85]. UML models assist a designer in analyzing relationships and dependencies in a system [91].

State Chart Diagrams

Statechart diagrams model a state machine which details the different states of a component in a system. They can be used to test an entire system or part of a system. A statechart describes the state machine by defining the state of an object and how the states are controlled by both internal and external events. Statechart diagrams are one of five types of UML diagrams that are used to model a system. Statecharts are typically used to model reactive systems by:

* Defining a state machine.
* Modeling the States of an object.
* Modeling the dynamic nature of a system.
* Modeling life of the system from beginning to end.

Statechart diagrams contain components including *states*, *events*, *transitions*, *parameters* and *guards*. The states have a hierarchical nature and may contain multiple states as well and *AND* or *OR* states. *Events* represent change in the model. *Transitions* define the previous and the next state. *Parameters* are used to input data into a state. *Guards* are Boolean conditions that allow or prevent states from transitioning. Offutt *et al.* [65] present a method to use UML statecharts to adapt pre-defined state based specification test data generation criteria to generate test cases. They also present several coverage criterion to transition coverage, full predicate coverage, transition pair coverage, and complete sequence coverage. They also present results from an empirical study. Due to the hierarchical nature of state chart diagrams, there is a need to flatten them to produce a sets of paths which are combined to form a test case. An automated approach is presented by Briand *et al.* [22]. They automate the process of generating test cases from a statechart using coverage criteria. Coverage criteria include:

* All Transitions.

* All Transition Pairs.

* Full Predicate.

* Round-Trip Paths.

By automating the process, they take each individual test path, identify the state, event, input parameters and transition needed for the test. They also introduce several methods for automating test generation. Lefticaru *et al.* [57] select test paths in a state machine, then use genetic algorithms (GA) to generate test data for them. The input parameters trigger the transitions. The GA locates the input parameters to satisfy a given requirement. The approach uses coverage criteria, identifies a test

path and input parameter values that would trigger the path. Prashanth *et al.* [3] present an efficient method to detect safety specification violations in dynamic behavior models of concurrent/reactive systems. Dynamic behavior is modeled as a UML statechart diagram. They also present a case study where the technique is applied to a generalized railroad crossing (GRC) system. The goal of the research is to reduce the number of states to be checked for detecting safety violations in the behavior of a reactive system. It is assumed that the system has multiple cooperating objects modeled by statecharts. They also present "relevant events", which is a set of events that can violate a safety property. A relevant events algorithm is described. The method is shown using a case study. The paper describes an event based algorithm for the verification of safety property violations in a UML statechart model of reactive systems [3].

Activity Diagrams

A UML Activity Diagram is used to understand the business process of the system [25]. It helps the developer understand the flow and the different paths of the software. Activity diagrams describe the dynamic elements in a system by representing data flow from one activity to the next. Activity Diagrams contain:

* Actions - Activities in a Modeled Process

* Initial State - Beginning of a Process or Workflow

* Decisions - Branching in the Data Flow

* Split/Join - Combines Concurrent Activities

* Final State- End of a Process or Workflow

UML activity diagrams and sequeuce diagrams have been used for generating test cases. Tripathy *et al.* [89] present an approach that transforms activity diagrams into activity graphs and sequence diagrams into sequence graphs. They then integrate the two graphs and traverse them to generate test cases.

UML Functional Testing

Kissoum *et al.* [52] present a paper about the importance of verifying the accuracy and reliability of software in multi-agent systems (MAS). They present automated test case generation for testing MAS. They also cover the formal specification of agent classes using the Maude algebraic language and describe the approach to test the agents and to generate test cases for specified MAS. They give examples of sequence and activity diagrams that specify the behavior of an agent [52].

Caire *et al.* [24] discuss agent behavioral representation which includes multi-agent behavior description (MABD) and single-agent behavior description (SABD). These agent behaviors drive the implementation phase. They discuss using UML activity diagrams with extensions. They also present a multi-level representation of agent behaviours and provide a new diagram called multi-agent zoomable behaviour description (MAZBD). They also present a multi-level approach to MAS testing which talks about agent level testing and the black-box behaviour of the agent. Their framework is also built on top of JADE [24].

UML diagrams as shown in activity diagrams or statechart diagrams can be used to test component based systems. Bertolino *et al.* [16] also combine sequence and state diagrams to generate a test model. Their test model is used to produce test cases.

UML Security Testing

UMLSec is an extension of UML which incorporates security properties such as secure flow of information, access control and confidentiality into a UML model. Jurjens *et al.* [47] use UMLsec to present security-relevant information in the UML diagrams in a system specification. They discuss distributed system security with respect to type of requirements (fair exchange, secrecy, secure information flow, secure communication link), UML extension mechanisms, outline of formal semantics, security analysis, and well-formedness rules. UML extension mechanisms expand UML metamodels by adding additonal elements or constraints to the UML models. A UML extension creates a profile using stereotypes, constraints and values. UMLSec covers object management group request for proposals (OMG RFPs) mandatory requirements (security, threat scenarios, security concepts, security mechanisms, security primitives, underlying physical security). They extend the work in [46] to present work for systematically testing security-critical systems. They specify the system with a formal language and generate test cases to identify security weaknesses. They present an approach using UMLSec and a general approach towards model-based security engineering. They also apply the approach to Common Electronic Purse Specification (CEPS). This approach works by adding formal security property semantics to UML models. As it offers a method for development and security analysis of a critical system, it does not answer our research questions.

Peralta *et al.* [68] present an approach to use UML to describe security characteristics such as design flaws in software. They present a set of UML stereotypes to identify behaviors that could compromise software security. The stereotypes are used to model sections of the software that could contain vulnerabilities. The approach has two goals identifying areas of vulnerability and generating security test cases.

This approach works by automatically generating test cases using the inserted security stereotypes. They then use the test cases to verify if a vulnerability could be possible in the test case. It does not answer our research questions as it has only been used on very simple test cases and further work is required to apply it to more elaborate test cases.

Wang *et al.* [95] presents a threat model-driven security testing approach for detecting undesirable threat behavior at run time. A threat model defines potential threats to a system. They are derived from design models that can extract threat traces. Each threat trace has an event sequence that should not occur during execution. Sequence diagrams are used to model the threat behaviors. A systematic approach is used that includes modeling threat scenarios with UML, deriving threat traces from the threat model, instrumenting the source code, monitoring the test execution driven by random test cases, and matching the execution traces with the threat traces. This approach works by extending model driven testing to security testing based on negative design specification. It does not address our research questions as we may not have access to the design models and the approach has also not been applied to testing large systems.

2.2.2 Finite State Machine (FSM)

Finite state machines (FSMs) are mathematical models used to design or test a computer system. An FSM contains states, transitions and triggering conditions. They have been used for generating test for many years [7]. They also support various types of coverage criteria including edge, node, edge-pair, prime path and W-method. The formal definition of a FSM is as follows [56]:

- M is a quintuple $M = (I, O, S, \delta, \lambda)$ where,
- I is a set of input symbols,
- O is a set of output symbols,
- S is a set of states,
- δ: $S \times I \rightarrow S$ is the state transition function,
- λ: $S \times I \rightarrow O$ is the output function, The machine starts in an initial state $s \in S$ and gets the input $i \in I$ it transitions to the next state with $\delta(s,i)$ and generates the output $\lambda(s,o)$ where $o \in O$.

FSM Functional Testing

Sabnani *et al.* [76] present a method to generate test sequences based on a deterministic FSM. They tour each state transition and assign a unique input/output (UIO) sequence to each state. Tsai *et al.* [90] demonstrate a method to automatically generate test cases for an object oriented class. They use a test case tree which is built from a state machine. Luo *et al.* [60] present an approach for test selection in distributed processes modeled in FSM. They also present a method to using synchronizable test sequences to automatically generate test sequences. Friedman *et al.* [36] generate tests based on FSM models. They present testing constraints and state and transition coverage criteria.

2.2.3 Extended Finite State Machine (EFSM)

Extended Finite State Machines (EFSMs) are an extension of an FSM. The FSM is extended to include variables and operations based on variable values. In FSMs, transitions are linked to inputs and outputs. However, in EFSMs, states transition or

fire when predicate conditions are satisfied and data operations are fully performed. When the transition is fired, the machine is moved from the current state to the next. The formal definition of a EFSM is as follows:

- An EFSM is a 5-tuple = (S, I, O, T, V) such that:
- S is a finite set of states,
- I is a set of input symbols,
- O is a set of output symbols,
- T is a set of transitions,
- V is a set of variables

However the transition $t \in T$ is a 6-tuple: $t = T(s_t, \prime s_t, i_t, o_t, P_t, A_t)$ such that:

- s_t is the current state,
- $\prime s_t$ is the next state,
- i_t is the input,
- o_t is the output,
- $P_t(\overrightarrow{v})$ is the predicate that must be true in order to execute the transition,
- $A_t(\overrightarrow{v})$ is the action on variable values,
- $(\overrightarrow{v})$ is the variable values

EFSM Functional Testing

EFSMs have a long history of being used for functional test and there has been alot of interest in using EFSMs to automate the test generation process [28]. Tahat *et al.* [84] propose an approach of requirement-based test generation. Their approach takes software specifications as individual requirements expressed in textual formats. They then automatically create a system model which is used to generate test cases. Their approach can then be extended to be used for regression testing. Derderian *et al.* [28] present an approach using a fitness function to recursively estimate the effort to find an input sequence to trigger a path through an EFSM. They then demonstate their approach using random sampling. Kalaji *et al.* [48] present an approach to solve the problems of generating tests with EFSMs. These problems include path feasibility and path test data generation. A path is considered not feasible if there is no input data to trigger the path. Due to the large amounts of data in these systems, discovering test data for a given path is also very difficult. Their approach utilizes optimization algorithms to test from EFSM models.

di Guglielmo *et al.* [29] present a functional automated test packet generation (ATPG) framework which uses the EFSM model to pseudo-deterministically generate a set of test sequences. A constraint solver is used to build test vectors that allow them to traverse the state space of the unit under test (UUT). The goal of the work is to determine test cases for faults that are difficult to locate.

EFSM Security Testing

Shu *et al.* [81] present a method using a learning-based approach to systematically and automatically test protocol implementation security properties. Test protocols are defined using a symbolic parameterized extended finite state machine (SP-EFSM)

model. They also include a message confidentiality security property. Black-box checking theory and a supervised learning algorithm are applied to examine the UUT while simulating the teacher with a conformance test generation scheme. To preserve black-box implementation, they maintain a model and update it as more information is learned using the test cases. The process continues until a security flaw is found or no new behaviors can be learned. This method is effective at identifying message confidentiality security properties. However, it is only focused on one security property and does not answer our research questions.

2.2.4 CESFM

Communicating Extended Finite State Machine (CEFSMs) [21] extended finite state machines (EFSMs) to include variables, operations based on variables, and interactions of variables. Traditional FSMs do not contain a communication mechanism and do not precisely model the interactions of systems [56]. In our case, we need to model at least two parallel processes, a functional process of the system under test and the attack process. They communicate at the rendezvous point or point of attack.

CEFSM's can be used to model and test distributed systems and network protocols [37]. CEFSMs are extended from EFSM to include a communication channel. CEFSM F is a tuple (E_F, C_F where an EFSM is represented as E_F and C_F is represented as a set of input and output communication channels). They have been used to model systems in [44][17][54]. CEFSM's consist of a finite set of EFSM's as well as communication channels between the EFSMs [56][37]. The benefit of CEFSMs is that it can model orthogonal states of a system in a flattened manner and does not need to create the entire system in a single state, such as in statecharts. CEFSMs are defined as a finite set of EFSMs as well as disjoint sets of input and output messages [56]. The formal definition of a CEFSM is as follows:

CEFSM = (S, s_0, E, P, T, M, V, A, C), such that:

- S is a finite set of states,
- s_0 is the initial state,
- E is a set of events,
- P is a set of boolean predicates,
- T is a set of transition functions such that T: S×P×E→S× A×M,
- M is a set of communicating messages,
- V is a set of variables,
- A is the set of actions, and
- C is the set of communication channels required in this CEFSM.

State changes (action language): The function T returns a next state, a set of output signals, and action list for each combination of a current state, an input signal, and a predicate. The function is defined as:

$T(s_i, p_i, get(m_i))/(s_j, A, send(m_{j_1},..., m_{j_k}))$ where,

- s_i is the current state,
- s_j is the next state,
- p_i is the predicate that must be true in order to execute the transition,
- $m_{i_1},..., m_{i_k}$ are the messages, and

The communicating message m_i is defined as:

(mId, e_j, mDestination) where,

- *mId* is the message identifier, and

- *mDestination* is the CEFSM the message is sent to.

An event e_i is defined as: *(eId, eOccurrence, eStatus)* where,

- *eId* is the event identifier that uniquely identifies it, and
- *eOccurrence* is set to false as long as the event has not occurred for the first time and to true otherwise, and
- *eStatus* is set to true when the event occurs and to false when it no longer applies. Note that *eStatus* allows reoccurring events to happen multiple times (loops in the model).

CEFSMs communicate by transferring messages through communication channels C that connect the output of one EFSM to the input of other EFSMs. Let C denote the set $\{\langle name, SYNC|ASYNC\rangle|$ for all the channels in the system$\}$ where *name* is the name of the communication channel and $SYNC$ and $ASYNC$ indicate that the channel is synchronous or asynchronous. A communication channel can be used differently according to different transitions. A channel c $\in$ C can be represented as $\langle name, t, get()/send()\rangle$ where, *name* is the name of the channel, $t \in$ T refers to the transition linked to this use of the channel, and *get()/send()* indicates whether this channel is an input or an output channel [37].

The action a_i may include an assignment and mathematical operation on the variables. The predicate is defined as a condition that must be met prior to the execution of a function. For example, *T(S_0, e_0, total = 4)/(S_1, {m_0, m_1}, (total = 0; increment(i)))* describes that if a CEFSM is in a state S_0 receives an event e_0 and the value of variable *total* is four at that time, it will move to the next state S_1 and output m_0 and m_1 after setting the *total* to zero and performing *increment(i)* [37]. The formal semantics are defined in [21].

CEFSM Functional Testing

CEFSMs can be used to model communicating processes [21]. Li *et al.* present a method to use CEFSM models to generate tests based on branching coverage [59]. Their method has two steps which include generating a flow diagram of the model and calculating the weight of each node in the diagram.

The weights can then be plotted to the matching branches in the CEFSM model. These priorities are used to ensure they get the most coverage with the fewest generated tests. They use a backward tracking method that selects the branches, then selects a variable value. The method uses priority levels to guide the test generation. They validate the feasibility of the transitions by forward checking. They prove that they can efficiently generate tests from CEFSMs. However, they do not address security concerns and do not address our research questions.

Bourhfir *et al.* present a method for automatically generating test cases and test sequences for a CEFSM model with asynchronous communication [18] [17]. Their approach works by incrementally computing a partial product and generating tests for only the transitions of CEFSMs which directly influence the behavior. This method has been applied to moderate sized systems. While this method demonstrates test generation from CEFSMs, it does not address the security issues in our research questions.

Bourhfir *et al.* present automated tools for testing CEFSM such as extended finite state machine test generator (EFTG) [19][32]. This method allows the user to create a complete reachability graph of the system which can be used for detecting inaccessible transitions. This method can be used to generate tests for full or partial testing of a system. While this method again demonstrates test generation from CEFSMs, it also does not address any of the security issues in our research questions.

Henniger *et al.* [43] discuss using asynchronous CEFSMs to create a test purpose description. They use a message sequence chart (MSC) to describe the intended behavior of the system. As this method focuses on modeling only the intentional behavior of the system, it does not address our research questions.

Hessel *et al.* [44] also present an algorithm for model-based generation of tests based on a priority driven reachability analysis. This method leverages knowledge about the total coverage found in the currently generated state space to direct and crop the remaining tree. They can then limit the test suite to a more reasonable size. This method does not answer our research questions as it is focused on testing the intended behavior of a system.

2.3 Security Testing

Security testing is the set of actions that are required to verify a software system's data and control flow are protected from malicious attacks. The primary rationale for testing the security of an embedded system is to identify and fix possible vulnerabilities. Basic security objectives are [50]:

* Confidentiality- Data remains private and is not disclosed to unauthorized sources.
* Integrity- Data is not altered as it is passed though the system.
* Authentication- The user can be identified.
* Authorization- The system can determine who is authorized to send or receive data.
* Availability- Data is available when it is needed.
* Non Repudiation- A guarantee that a transaction or a communication occurred.

2.3.1 Security Testing Process

Due to higher levels of software intensive systems and privacy concerns, the need for information security is very important [50]. To define security in a system, it must be understood what assets need to be protected and the cost of protecting them [63]. Traditional software testing uncovers the occurrence of errors, but not necessarily the absence of features that would protect it [85] [63]. Security testing should incorporate the testing of security processes with traditional software testing to guarantee that software executes correctly even when it is being attacked maliciously [63]. Security testing is very difficult because the tester must think like an attacker to compose the system's architecture with an attacker's behaviors to adequately secure a system [63] [91]. Traditional security testing involves running a series of known tests that had been known to exploit a system [85]. While effective, this method identifies known vulnerabilities but does nothing to uncover new vulnerabilities [85]. Security testing can be performed in the following steps [85]:

* Discovery - Identify the elements of a system
* Vulnerability Scan - Use automated tools to identify security issues
* Vulnerability Assessment - Use vulnerability scans and discovery identify potential vulnerabilities in system.
* Security Assessment - Build a description of the system security.
* Penetration Test - Attempt to simulate a malicious attack on the system.
* Security Audit - Provide an assessment of the system's security using information gathered
* Security Review - Verify the system meets industry standards for security

There are several traditional approaches to testing software security [91]. These approaches include security checklists, common tools (i.e. monitoring or interfacing tools), fuzzing (blasting a system with random input), vulnerability scanners, security use cases (converting standard use cases by changing input values), hacking (employing hackers to look for vulnerabilities) and model based security test (MBST). Our research questions, focus MBST. We are demonstrating a method to use a behavioral tests, mitigation test and attack scenarios to build a systematic approach to security testing (MBST).

2.3.2 Vulnerabilities

Software vulnerabilities are areas in software that an attacker can manipulate [63]. Understanding vulnerabilities is a difficult task. Many of them have been identified, named and renamed over the years [91]. Threats to systems include: evolution and growing elegance of bot armies, financially influenced hacking, sophistication of malware and increased use of virtual machine root kits [83].

Once single vulnerabilities have been identified, they should be categorized into groups of similar threats [91]. Testing can then be focused on common threats. Categorizing and prioritizing tests can be accomplished by understanding the system's environment and software architecture. After vulnerabilities have been understood, testers need to understand security testing. If there is a security issue in a single component that has been integrated into a system, it can cause vulnerabilities for the entire system. In general, software components are full of vulnerabilities and difficult to debug at the source code level [97]. Security risks can be prioritized as follows: urgent (data cannot be recovered or system crashes), high (critical functions do not perform properly), medium (critical functions can be performed with a work around), low (frustration or nuisance) or none (no issues) [85].

Design and Code Vulnerabilities

Poor coding practices are the cause of many vulnerabilities [50]. They include code implementation errors (i.e. buffer overflows), interprocedural errors (i.e. race conditions), or design level errors (i.e. error handling) [63]. Vulnerabilities can usually be divided into two groups: defects at the implementation level or defects at the design level. Design level vulnerabilities are very difficult to locate. Design level flaws could include issues in object oriented systems with error handling or data sharing issues, timing issues or data transfer issues. Each design level flaw in a program can be exploited [63]. A risk assessment analysis can be performed byd eveloping security abuse cases, creating security requirements, performing architectural risk analysis, developing test plans, implementing analysis tools, executing security tests, running penetration testing in the final environment and cleaning up any security issues. Code faults can generally by categorized into four groups dependency issues, user input, design issues and implementation issues [85]. Dependency Issues are identified when software interfaces with multiple applications that subsequently load remote libraries. The software will inherit any vulnerabilities that exist in any of the dependent files. Testers must inspect each dependency and verify its vulnerabilities. This may be difficult if source code is unavailable. User input vulnerabilities are introduced when memory buffer overflows can occur when input is keyed in by a user. If extra data is entered, these values can be seen as instructions and executed. This can cause erratic behavior and can be difficult to debug. Also, particular character string combinations could be executed as a command. Design issues are vulnerabilities introduced during the software design phase. During design, testing ports could be inadvertently left open, insecure default values left in place, or accesses to functions left open. These issues are often put in place for testing and then forgotten. These gaps can grant an

attacker critical access to a system. Implementation issues occur when secure designs are implemented in a way that is insecure. An example of this is a man-in-the-middle attack. This attack occurs when an attacker can access data between the time it is accessed and utilized. The best way to avert this attack is to check the time the data was accessed and if it exceeds some nominal value simply retrieve the data again.

Three examples of known vulnerabilities that may only be detected with black box testing are side effect behavior, control path vulnerabilities and issues in COTS components. Side effect behavior occurs when software can have tasks executing that are not part of the requirements. These additional undefined tasks may have been used for early dubugging and may not be detected by automated testing [85]. An example of an additional task vulnerability is the Windows RDISK task. This function is intended to create an emergency repair disk for a computer. However, it also generates a temporary file that contains the system's configuration sensitive information with full read permissions [85]. Additional task vulnerabilities are sometimes a result of poor coding practices. Tests to determine the function's correctness, often do not uncover extra features unless the system were to crash while being tested. Control path vulnerabilities occur when a software system is transferring data or control and it is possible to alter the execution paths. Contemporary software systems are a collaboration of software components and operating systems which depend on data and control transfer from each other [97]. Therefore, even from the outside of the system, its activity can be monitored. COTS Components may come from many different vendors and from all over the world. Each component could pose a security risk for the entire system [83].

2.3.3 Defining Security Requirements

An important activity in an evaluation of information security is the generation of security objectives and requirements. These requirements are based on an analysis of risks, threats and vulnerabilities [50]. Security requirements are considered [50]:

- Functional Requirement - the system must perform a specific task that is testable. A test case demonstrates the requirement and provides a pass or fail result [50].

- Non-Functional Requirement - defines how the software will perform a specific task.

- Positive Requirement - designates that the system must do some particular task, not necessarily a security requirement [50].

- Negative Requirement - directs that a particular task never occur in the system.

2.3.4 Testing Tools

Testing Tools help testers identify and distinguish security flaws [85]. To test an existing or create a new tool, the following items must be kept in mind [91]:

- Does the tool focus on the proper issue (e.g. buffer overflows)?

- Does the tool work with various systems and does not require complete knowledge of these systems?

- Does the tool support the development and test life cycle of the system?

- Are the results valuable?

While some tools are automated, others depend upon highly skilled security personnel to use them [91]. Automated tools can test a large number of cases very quickly, human creativity is also an invaluable tool in preventing attacks [91]. Two examples of tools that are available are:

* Attack Trees - Attack trees are used to model possible attacks on a system. They are successful at modeling security issues. However, there is no methodical way to generate them.

* STRIDE method - The STRIDE method (spoofing, tampering, repudiation, information disclosure, denial of service and elevation of privilege) takes a predefined list of known attacks and evaluates how a system will react to them [91].

Better tools are needed to keep up with the demands of security. Ideas for other tools include development level tests, advanced threat modeling and improved security debugging tools [91]. The goal of these tools should be to increase the level of assurance the tester and the community have in a software system [91].

2.3.5 Measuring Security

Methods of measuring security are sometimes used to create metrics that detail the security of a system [50]. These metrics are not precise or concrete but provide some insight into a system's security. Direct testing is a method that performs functional or penetration testing. An evaluation method is completed by assessing known criteria. An assessment can be performed by analyzing risks threats and vulnerabilities. Accreditation occurs when a system is proven in a specific environment. Training can support measuring security by educating security personnel. Finally, an observation can be performed by witnessing a system avert threats [50].

2.3.6 Relevant Papers

There are several existing approaches for security testing software. We divide the papers into model and non model based as well as testing and analysis. The approaches are generally good at identifying where attacks could occur in a system. However, they are not applicable in large systems and do nothing to test proper mitigations of attacks.

Table 2.2: Security Research Matrix

	Model	Non-Model
Test	[95] [61] [77] [79] [34] [41] [64] [46] [20] [99] [12]	[86] [87] [63]
Analysis	[47] [13]	[10] [88]

Model Based Security Test

Gu *et al.* [87] discuss general security tests. They discuss formal security testing, model-based security testing, fault injection-based testing, fuzzy testing, vulnerability scanning, property-based, white box-based, risk-based security tests and taxonomy/function of security testing tools. This paper gives a brief overview of security testing techniques. However, they don't give a specific method.

Wang *et al.* [95] demonstrate an approach using sequence diagrams to detect threat behavior at run time. Their approach models threat scenarios with UML, derives threat traces from the threat model and instruments the source code. Finally, they monitor the test execution driven by random test cases and match threat traces.

Marback *et al.* [61] proposes a threat model-based security testing approach that automatically generates security test sequences from threat trees and transforms them into executable tests. Their approach has three main goals building threat models with threat trees, generating security test sequences from threat trees and creating executable test cases by considering valid and invalid inputs.

The authors define threat modeling as a systematic process of identifying, analyzing, documenting, and mitigating security threats to a software system. Their approach identifies the assets of an application. They then determine and rank the threats to an application. Lastly, they mitigate the threats.

Groundwork Demonstration

We demonstrate this approach by using a simple sniff attack to gather data on a wireless Ethernet connection. We follow the steps of the approach. In step 1, we build a threat model with a threat tree shown in Figure 2.2. In step 2, we generate a security test sequences from our threat trees.

Our test sequences are as follows:

Sequence 1: SniffDataBus.PassiveSniffing.DownloadnetSlumber.
DisplaySignals

Sequence 2: SniffDataBus.PassiveSniffing.DownloadKismet.DisplaySSIDs

Sequence 3: SniffDataBus.PassiveSniffing.DownloadKismet.CollectMac

Sequence 4: SniffDataBus.PassiveSniffing.MineSSID.DownloadKismet.
DisplaySSIDs

Sequence 5: SniffDataBus.PassiveSniffing.MineSSID.DownloadKismet.
CollectMac

Sequence 6: SniffDataBus.PassiveSniffing.MineSSID.AttachNetwork.
DownloadAirsnort.SniffWep

Sequence 7: SniffDataBus.PassiveSniffing.MineSSID.AttachNetwork.
DownloadCowPatty.SniffWep

Sequence 8: SniffDataBus.PassiveSniffing.MineSSID.AttachNetwork.
DownloadASLeap.CollectAuthData

Sequence 9: SniffDataBus.PassiveSniffing.MineSSID.AttachNetwork.CollectData
.DownloadWireshark.ScanPacket

We follow step 3 to create executable test cases. Tests are generated by considering both valid and invalid inputs and incoporating operations, variables and passing messages. These test cases demonstrate how the method would travesse through the attack tree and mitigate the attack. While effecitve, it should be noted that the number of test cases is very high for this simple example.

Test 1: SniffDataBus[true].PassiveSniffing[true].DownloadnetSlumber[true]
.DisplaySignals[true]

Test 2: SniffDataBus[false].Return[error]

Test 3: SniffDataBus[true].PassiveSniffing[false].Return[error]

Test 4: SniffDataBus[true].PassiveSniffing[true].DownloadnetSlumber[false]
.Return[error]

Test 5: SniffDataBus[true].PassiveSniffing[true].
DownloadnetSlumber[true].DisplaySignals[false].Return[error]

Test 6: SniffDataBus[true].PassiveSniffing[true].DownloadKismet[true].
DisplaySSIDs[true]

Test 7: SniffDataBus[false].Return[error]

Test 8: SniffDataBus[true].PassiveSniffing[false].Return[error]

Test 9: SniffDataBus[true].PassiveSniffing[true].DownloadKismet[false]
.Return[error]

Test 10: SniffDataBus[true].PassiveSniffing[true].DownloadKismet[true].

DisplaySSIDs[false].Return[error]

Test 11: SniffDataBus[true].PassiveSniffing[true].DownloadKismet[true]
.DisplaySSIDs[true]

Test 12: SniffDataBus[false].Return[error]

Test 13: SniffDataBus[true].PassiveSniffing[false].Return[error]

Test 14: SniffDataBus[true].PassiveSniffing[true].DownloadKismet[false].
Return[error]

Test 15: SniffDataBus[true].PassiveSniffing[true].DownloadKismet[true].
DisplaySSIDs[false].Return[error]

Test 16: SniffDataBus[true].PassiveSniffing[true].DownloadKismet[true].
DisplaySSIDs[true]

Test 17: SniffDataBus[false].Return[error]

Test 18: SniffDataBus[true].PassiveSniffing[false].Return[error]

Test 19: SniffDataBus[true].PassiveSniffing[true].
DownloadKismet[false].Return[error]

Test 20: SniffDataBus[true].PassiveSniffing[true].
DownloadKismet[true].DisplaySSIDs[false].Return[error]

Test 21: SniffDataBus[true].PassiveSniffing[true].
DownloadKismet[true].CollectMac[true]

Test 22: SniffDataBus[false].Return[error]

Test 23: SniffDataBus[true].PassiveSniffing[false].Return[error]

Test 24: SniffDataBus[true].PassiveSniffing[true].DownloadKismet[false].
Return[error]

Test 25: SniffDataBus[true].PassiveSniffing[true].DownloadKismet[true].
CollectMac[false].Return[error]

Test 26: SniffDataBus[true].PassiveSniffing[true].DownloadKismet[true].

CollectMac[true]

Test 27: SniffDataBus[false].Return[error]

Test 28: SniffDataBus[true].PassiveSniffing[false].Return[error]

Test 29: SniffDataBus[true].PassiveSniffing[true].DownloadKismet[false].
Return[error]

Test 30: SniffDataBus[true].PassiveSniffing[true].DownloadKismet[true].
CollectMac[false].Return[error]

Test 31: SniffDataBus[true].PassiveSniffing[true].MineSSID[true].
DownloadKismet[true].DisplaySSIDs[true]

Test 32: SniffDataBus[false].Return[error]

Test 33: SniffDataBus[true].PassiveSniffing[false].Return[error]

Test 34: SniffDataBus[true].PassiveSniffing[true].MineSSID[false].
Return[error]

Test 35: SniffDataBus[true].PassiveSniffing[true].MineSSID[true].
DownloadKismet[false].Return[error]

Test 36: SniffDataBus[true].PassiveSniffing[true].MineSSID[true].
DownloadKismet[true].DisplaySSIDs[false].Return[error]

Test 37: SniffDataBus[true].PassiveSniffing[true].MineSSID[true].
DownloadKismet[true].CollectMac[true]

Test 38: SniffDataBus[false].Return[error]

Test 39: SniffDataBus[true].PassiveSniffing[false].Return[error]

Test 40: SniffDataBus[true].PassiveSniffing[true].MineSSID[false].
Return[error]

Test 41: SniffDataBus[true].PassiveSniffing[true].MineSSID[true].
DownloadKismet[false].Return[error]

Test 42: SniffDataBus[true].PassiveSniffing[true].MineSSID[true].

DownloadKismet[true].CollectMac[false].Return[error]

Test 43: SniffDataBus[true].PassiveSniffing[true].MineSSID[true]
.AttachNetwork[true].DownloadAirsnort[true].SniffWep[true]

Test 44: SniffDataBus[false]..Return[error]

Test 45: SniffDataBus[true].PassiveSniffing[false].Return[error]

Test 46: SniffDataBus[true].PassiveSniffing[true].MineSSID[false]
.Return[error]

Test 47: SniffDataBus[true].PassiveSniffing[true].MineSSID[true].
AttachNetwork[false].Return[error]

Test 48: SniffDataBus[true].PassiveSniffing[true].MineSSID[true].
AttachNetwork[true].DownloadAirsnort[false].Return[error]

Test 49: SniffDataBus[true].PassiveSniffing[true].MineSSID[true].
AttachNetwork[true].DownloadAirsnort[true].SniffWep[false].
Return[error]

Test 50: SniffDataBus[true].PassiveSniffing[true].MineSSID[true].
AttachNetwork[true]
.DownloadCowPatty[true].SniffWep[true]

Test 51: SniffDataBus[false].Return[error]

Test 52: SniffDataBus[true].PassiveSniffing[false].Return[error]

Test 53:SniffDataBus[true].PassiveSniffing[true].MineSSID[false].
Return[error]

Test 54: SniffDataBus[true].PassiveSniffing[true].MineSSID[true].
AttachNetwork[false].Return[error]

Test 55: SniffDataBus[true].PassiveSniffing[true].MineSSID[true].
AttachNetwork[true].DownloadCowPatty[false].Return[error]

Test 56: SniffDataBus[true].PassiveSniffing[true].MineSSID[true].

AttachNetwork[true].DownloadCowPatty[true].SniffWep[false].
Return[error]

Test 57: SniffDataBus[true].PassiveSniffing[true].MineSSID[true]
.AttachNetwork[true].DownloadASLeap[true].CollectAuthData[true]

Test 58: SniffDataBus[false].Return[error]

Test 59: SniffDataBus[true].PassiveSniffing[false].Return[error]

Test 60: SniffDataBus[true].PassiveSniffing[true].MineSSID[false]
.Return[error]

Test 61: SniffDataBus[true].PassiveSniffing[true].MineSSID[true]
.AttachNetwork[false].Return[error]

Test 62: SniffDataBus[true].PassiveSniffing[true].MineSSID[true]
.AttachNetwork[true].DownloadASLeap[false].Return[error]

Test 63: SniffDataBus[true].PassiveSniffing[true].MineSSID[true]
.AttachNetwork[true].DownloadASLeap[true].CollectAuthData[false]
.Return[error]

Test 64: SniffDataBus[true].PassiveSniffing[true].MineSSID[true]
.AttachNetwork[true].CollectData[true].DownloadWireshark[true]
.ScanPacket[true]

Test 65: SniffDataBus[false].Return[error]

Test 66: SniffDataBus[true].PassiveSniffing[false].Return[error]

Test 67: SniffDataBus[true].PassiveSniffing[true].MineSSID[false]
.Return[error]

Test 68: SniffDataBus[true].PassiveSniffing[true].MineSSID[true]
.AttachNetwork[false].Return[error]

Test 69: SniffDataBus[true].PassiveSniffing[true].MineSSID[true]
.AttachNetwork[true].CollectData[false].Return[error]

Test 70: SniffDataBus[true].PassiveSniffing[true].MineSSID[true]
.AttachNetwork[true].CollectData[true].DownloadWireshark[false]
.Return[error]
Test 71: SniffDataBus[true].PassiveSniffing[true].MineSSID[true]
.AttachNetwork[true].CollectData[true].DownloadWireshark[true].
ScanPacket[false].Return[error]

Our example for a simple sniff attack takes 71 executable tests. If we have 5 attacks we would need attack trees to cover at least 71 steps at each interface. This approach is can identify vulnerabilities in software systems. However, the approach is not scalable to large systems. We will compare this approach with ours in section 4.

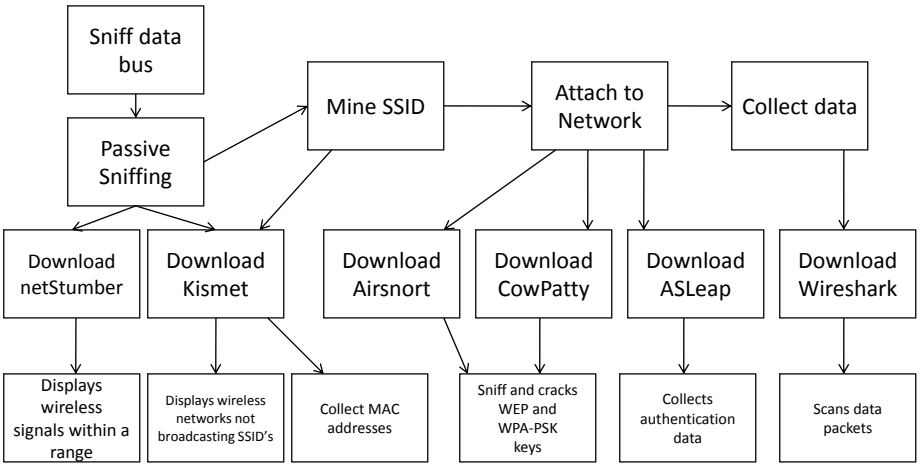

Figure 2.2: Marback Approach Example

Salas *et al.* [77] Present a three-model framework: model of app, model of implementation, and model of attacker. Their approach outlines some of the relevant techniques in model based and security testing. They further discuss vulnerability analysis, fault-model-based testing, model-based security vulnerability testing and test case generation. They also identify how the use of each model reveals certain problems such as under-specification problems in security vulnerabilities.

Schieferdecker *et al.* [79] focus on model-based security testing. They cover the types of models in the model based security test (MBST) including architectural and functional models, threat, fault, risk models, and weakness and vulnerabilities models. They also review activities involved in MBST which includes identifying security test objectives and methods, designing functional test models, determining test generation criteria, generating the tests, and assessing the test results. They also present approaches in the DIAMONDS project which focuses on risk-based MBST and model-based fuzz testing. Felderer *et al.* [34] present the value of MBT and the risks associated with automated test generation and the value of MBT. Security testing verifies security requirements: confidentiality, integrity, authentication, authorization, availability, and non-repudiation.

Mouelhi *et al.* [64] presents a model-driven approach for specifying, deploying and testing security policies in Java applications. They present a generic security metamodel, define the formalism and then define a policy according to this formalism. Their approach builds the security model for the application, produces specific policy decision points, connects the security framework with the functional code of the application, systematically use access control policy (ACP) to enforce the policy enforcement point (PEP) and uses mutation testing to ensure that final running code conforms to the security model. These five steps are performed to enforce security policies on applications.

Jurjens *et al.* [46] presents work for systematically testing security-critical systems. They cover a general approach towards model-based security engineering. Their approach uses UMLsec with regards to common electronic purse specification (CEPS) commonly used by banks and credit cards such as VISA International. Bozic *et al.* [20] Present an approach to use model-based fuzz testing as well as mutation testing and inference assisted evolutionary fuzz testing. They demonstrate their approach with SQL injection. This approach uses model based test for security testing, however it does not incorporate mitigations as our research questions require.

Wimmel *et al.* [99] presents an approach to generate test sequences for transaction systems from a formal security model supported by computer aided software engineering (CASE) and AutoFocus tools. They compare security models in CASE and AutoFocus tools with UML component diagrams. They also cover vulnerability coverage using mutations and attack scenarios and concretization of abstract tests. This paper presents a method with a formal security model and vulnerability coverage, but it does not address our research questions.

Barnum *et al.* [12] present the need to have a firm grasp of the attacker's perspective and the approaches used to exploit software. They note how each attack pattern can give a list of information, such as attack prerequisites, method, skill, weaknesses, solutions and mitigations. They also cover the concept, generation, and usage of attack patterns as a knowledge tool in the design, development, and deployment of secure software. Their method does not address modeling the intended behavior as required in our research questions.

He *et al.* [41] present a key paper related to this dissertation. They present an attack scenario based approach that can be used to secure software at the design stage. Attack paths are used to automatically create security test cases. Security test cases can then be used to verify a system's level of security and its ability to

defend itself against attacks [41]. Software security requirements increase with the amount of critical data the system processes. Testing for security at the design phase improves its chances to protect itself against vulnerabilities. The method is detailed below [41]:

* System components are modeled by extending the activity diagram EAD (Extended Activity Diagram) to include trust levels and boundaries- Threats can arise from data being transferred from a component with a low level of security to a component with a higher level of security without proper security measures. The method uses the following definitions:
    * Definition 1: Trust level is the measure of trust about a module in a system. Let TL= { low, medium, high, unknown}
    * Definition 2: Trust Boundary is a virtual or physical boundary of trust levels. Let TB = trust boundary
* System threats are modeled using NUTM (New Unified Threat Model)- Threat modeling has traditionally used a threat tree based UTM (Unified Threat Model). A UTM node can be a goal that a set of subnodes satisfy. It could also be a set of subnodes who each have the goal of satisfying the node. NUTM expands UTM to model a node of UTM as well as a complex attack path [41].
    * Definition 3: The attack sequence is a sequence of nodes of the unified threat model. Let AS=$<N_1, N_2, \ldots N_n>$represent the attack sequence.
    * Definition 4: The complex attack is an attack path that includes sequential and concurrent relations between the attack sequences Let cPa=$\{ <N_1, N_2, \ldots N_n>, \ldots, <N_1, N_2, \ldots N_n>\}$ represents the complex attack path.

    * Definition 5: The attack scenario identifies how an attacker will exploit a system and identifies mitigations to prevent the attack.

* Attacks are created and verified- Preventing attacks can be better achieved by reviewing the attacker's goals, entry points and reasons for exploiting a system. Attacks are created as an attack scenario that encapsulates the target of the attack (EAD object nodes), the system's actors (or activity section), a feasible attacker (NUTM's root node) and a possible mitigation node. The EAD object nodes can be used to identify the TB of data paths. These are possible attacker entry points [41].

* Test cases are generated with the following items [41]:

    - Test Case ID and Name (Used for identification of test)
    - Precondition (A set of conditions that must be satisfied)
    - Security Reaction (System Response to prevent attack)

  Test criteria require that tests must exert each node and relation and cover each attack path in the NUTM at least once . Test cases can then be executed on the system and a tester can determine if the system is successful at preventing the attack [41].

* System threats are mitigated- If the system cannot successfully prevent the attack from occurring, it must be improved to meet the security requirements. Attacks should be prevented by disabling the attacker's access.

This approach can help developers improve a system's design by identifying possible areas for attack as well as the system's response to the attack [41]. They define a set of test requirements as TR. They define the test set as T and the coverage

criteria C. The coverage criteria can be satisfied if and only if at least one T exists for each TR. Where, TR equals the attack scenario and complex attack path. Tests are generated by defining sequential and concurrent attacks in an attack path.

Groundwork Demonstration

We apply this approach to our simple robotics example and give a single attack to demonstrate their approach as shown in Figure 2.3. In our example we show a UML diagram for a robot with the following functions:

* Power up Interface- provides the mechanism to power the robot on
* Robot Power On - Indicator that the power is on
* Robot Start Video - the interface that starts, stops and sends the robot's camera data
* Robot Read Position - Sends the robots current location
* Robot Wait for Position - the interface that accepts and executes commands to move the robot

A trust boundary line is drawn between the areas where attacks can occur. The functions above the trust boundary line are considered to not be high risk interfaces. Functions below the trust boundary are considered to be high risk. As each interface needs to be tested with each attack, the state space becomes very large, very quickly. Test Case ID: STC 01

- Name: DOS RFJam
- Precondition: Robot is functional and powered on

- Test Steps:
    - Attacker Action: Jam robot with RF signals
    - System expected reaction: Detect this action, turn off RF communication channels, return robot to safe position
- Expected Security Reaction: The system should prevent attacks from jamming robot with RF signals

Test Case ID: STC 02

- Name: DOS Flood
- Precondition: Robot is functional and powered on
- Test Steps:
    - Attacker Action: Flood robot with commands
    - System expected reaction: Detect this action, turn off communication channels, return robot to safe position
- Expected Security Reaction: The system should prevent attacks from flooding robot with commands signals

Test Case ID: STC 03

- Name: Sniff and Spoof
- Precondition: Robot is functional and powered on
- Test Steps:
    - Attacker Action: Gather robot communication data

        - System expected reaction: Detect this action, log message, send message to controller
        - Attacker Action: Create and send malicious commands
        - System expected reaction: Detect this action, stop listening to commands and return robot to safe position
    - Expected Security Reaction: The system should prevent attackers from listening to data or acting on malicious commands

Test Case ID: STC 04

- Name: Sniff and Man in the Middle (MITM)
- Precondition: Robot is functional and powered on
- Test Steps:
    - Attacker Action: Gather robot communication data
    - System expected reaction: Detect this action, log message, send message to controller
    - Attacker Action: Pass data on to intended recipient
    - System expected reaction: Detect this action, log message, send message to controller
    - Attacker Action: Replace instructions
    - System expected reaction: Detect this action, stop listening to commands and return robot to safe position
- Expected Security Reaction: The system should prevent MITM attacks

Test Case ID: STC 05

- Name: GPS Jam
- Precondition: Robot is functional and powered on
- Test Steps:
    - Attacker Action: Jam robot with miscellaneous GPS signals
    - System expected reaction: Detect this action, turn off GPS communication channels, return robot to safe position
- Expected Security Reaction: The system should prevent attacks from jamming robot with GPS signals

In the example, a simple robot with 5 interfaces and 5 attacks requires 25 tests. The approach is not scalable for large systems. We will compare it with our method in section 4.

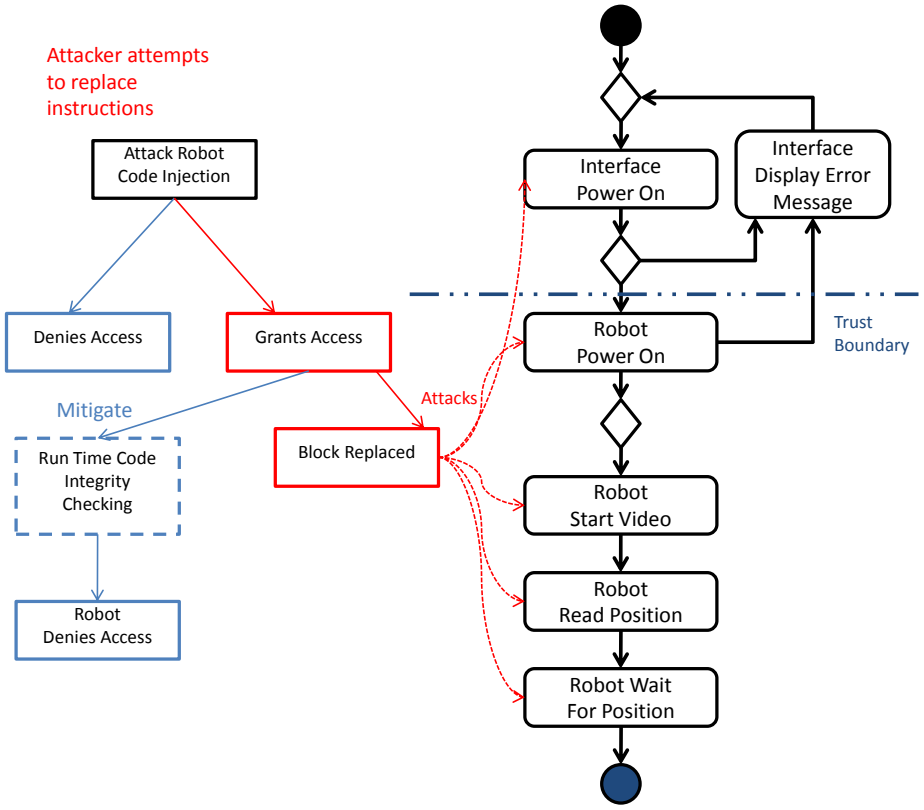

Figure 2.3: He Approach Example

Non-Model Based Security Test

Thompson *et al.* [86] present a paper that discusses black box testing in all operating conditions. They discuss how extreme conditions (stress on system, network failure, I/O error, memory failure) can affect the security of a system. They also discuss the need to integrate these failures into test cases. They also highlight the fact that error handling is generally not testing as thoroughly as the main functional code, leaving any code untested in danger and that all code should be executed and tested under adversarial conditions.

McGraw *et al.* [63] focus on risk-based security tests. They discuss how classical black-box testing is not enough for security testing. They also reiterate the notion that coding errors create most of the vulnerabilities in software. Their definition of software attacks includes timing attacks, race conditions, and two-stage buffer overflow attacks.

Antoniol *et al.* [10] present a paper on privilege escalation testing, SQL injection, robustness (fuzz) and penetration testing, and intrusion detection systems (IDS). They define vulnerabilities and examine search based software testing (SBST) approaches to detect them. They also discuss genetic programming and using trees to model IDS. Tondel *et al.* [88] present an approach to develop a vulnerability repository. The repository would include information in all software development phases (requirements, design, implementation and testing). Each record in the repository would address the following info: date, category, where, root cause, risk and countermeasure.

Stytz et al. introduce the need for an intelligent system to secure software systems. An intelligent system can be used to test software for defects throughout the development cycle [83]. It should be able to reveal software flaws and provide information

on assessing the system's environment [83]. The intelligent system would have the capability of executing various types of attacks, execute more than one attack at a time, provide an assessment of a system's security at the component level and report security metrics [83]. To design an intelligent system the user must ascertain the intelligent testing system requirements, construe its behaviors by using use case diagrams and documentation, develop software that meets requirements with use case diagrams and documentation and refine the system. This cycle is recursive until the testing system has reached an acceptable performance and defects are revealed [83]. The development knowledge base must be modified with each system tested and threats to the system. To simulate a real attack, each threat case should contain a narrative, and be modeled using a UML use-case and state chart diagrams [83]. The authors have not yet built an intelligent system. However, they claim that it should be able to improve the security of software and divert attacks. It also proves that there is a need for a solution to secure software.

Attack Simulator

Beech *et al.* [14] present their design to implement and evaluate an attack simulator tool. This tool can be used to dynamically inject well known attacks into a system without changing the original source code. Their simulator builds test cases using a tuple which includes:

- A compiled program P

- Protection Mechanism

- Input to P

The attack simulator will insert attacks at different points and evaluate how the security mechanism reacts. The simulator injects machine instructions. Therefore,

the integrity of the compiled program is never changed. The simulator is built using a dynamic compiler. The dynamic compiler goes into a loop where it builds machine instructions that would execute sequentially. The requirements for the simulator are as follows:

- Generality- Support multiple languages and vulnerabilities or attacks

- Systematic testing - Inject attacks at any point

- Automatic - Minimal User specifications required

- Robustness - Accurate reporting of results

- Low overhead - Must not incur overhead to the program

Their simulator is successful at automatically identifying attack points reliably with minimal overhead. They are also successful at identifying a common type of attack called stack smashing. While their simulator is successful in identifying attack points, it is a much different approach than ours which weaves in the mitigations at the attack points. We could, however, make use of the simulator to execute our security test cases.

2.4 Mitigations

Security mitigations are s series of events that are intended to keep a computer system safe against a malicious attack. Mitigations can be pattern based, in which they follow a specific arrangement. They can also be non-pattern based , in which they do not follow a particular model. This dissertation focuses on pattern based mitigations.

Table 2.3: Mitigation Research Matrix

Mitigation Research Matrix		
	Pattern	Non-Pattern
Security	[98] [82] [12] [38] [5] [6] [27]	[35]

2.4.1 Pattern Based Mitigations

Wiesauer *et al.* [98] present security patterns including: secure pipe, secure logger, single access point, check pointed system, and limited view patterns. They present many different security design pattern definitions and specifications as well as different taxonomies for security design patterns. They argue that current methods are inadequate and present a new taxonomy based on attack patterns. Their attack patterns include, spoofing, identity spoofing, content spoofing, man in the middle, denial of service, resource depletion, privilege exploitation, etc. They also explore the relationship between attack patterns, security design patterns and test cases. They also measure the quality of security design patterns by analyzing documentation and repeatability. Security design patterns must have adequate documentation to allow designers who are not security experts to use them. Test cases are used to demonstrate repeatability by showing that patterns can be applied correctly, and attacks will be prevented.

Smith *et al.* [82] also research security test patterns similar to how software design patterns work; a universally accepted way of designing software in certain situations. A security test pattern is a template of a test case that exposes vulnerabilities, typically by emulating what an attacker would do to exploit those vulnerabilities. This

article demonstrates the use of security test patterns to generate better black box security tests. They demonstrate their technique by conducting a user study of 21 graduate and 26 undergraduate students with a low level of security testing knowledge. The students used a tool to generate a black box security test plan using security test patterns. The results of the case study indicated that by using valid security test patterns, students with little experience could effectively generate black box security test plans.

Barnum *et al.* [12] discusses the need to have a firm grasp of the attacker's perspective and the approaches used to exploit software. Attack patterns can help understand the types of exploits and attacks. They can be used to build secure software as well as define mitigation patterns. Each attack contains:

- Pattern Name and Classification - unique identifier
- Attack Prerequisites - Required conditions for the attack to be successful
- Description - sequence of events in an attack
- Related Vulnerabilities or Weaknesses - weaknesses that the attack will leverage
- Method of Attack - type of attack used
- Attack Motivation or Consequence - The goal of the attacker
- Attacker Skill - Specific knowledge of the attacker
- Resources - Computing resources required for attack
- Solutions and Mitigations - Any steps that can mitigate the attack
- Context Description - Describes the context on a successful attack

Gegick *et al.* [38] discuss how attack patterns can show security vulnerabilities in a software system design. They also present how to match attack patterns with vulnerabilities in the design phase to increase security efforts early on in the software development process. This article is focused on how to build secure software by integrating security early on in the design phase. They also discuss about being able to use external intrusion detection or firewalls. They analyze security vulnerabilities in four vulnerability databases to determine which ones are valid and decide common methods to exploit them. They present an analysis of components that can be used to exploit a vulnerability. They use regular expressions to abstract and formalize events that can be used in an attack. This research demonstrates that security vulnerabilities can be identified by using abstract attack patterns. The technique of finding vulnerabilities can be accomplished by matching a sequence of components in a system design. This method can be used to increase security awareness during the software life cycle. They also perform a study to compare which vulnerabilities they can identify and how to best increase security awareness.

Alvi *et al.* [5] proposes a method to classify software security patterns. Their approach is unique in using security patterns to prevent the cause of an attack on a system. They present a security pattern template based on analyzing available pattern templates. They then incorporate classification parameters. They also present a classification for software security patterns using software development life cycle (SDLC) phases such as requriements, design, code, test and deployment . Each life cycle phase is further classified by using security flaws, requirement security objectives , design security properties, and implementation attack patterns. Their method demonstrates how a user can map the attack pattern to a security pattern and select the best option. Alvi *et al.* [6] also present further work to classify the different types of security patterns based on attack patterns and on security flaws.

de Muijnck-Hughes *et al.* [27] discuss security design patterns, for example, solutions to recurring security problems in software. They discuss issues affecting security design patterns (how security problems can be addressed). They note that there are a variety of existing pattern templates as shown in Table 2.4 can be used by pattern developers. However, there is no central repository of pattern templates. Some of these pattern templates are unique, while others are variations of existing templates. Due to the large number of pattern templates and no central repository, accessibility of these patterns to developers is limited. They recommend that, patterns should be put in a formally evaluated framework. Mitigations can be derived using software development experience to locate recurring problems and their associated solutions [103].

Security Pattern Catalog	
Name	Goal
Single Access Point	Provide a security module and access to login
Check Point	Consolidating security checks and impacts
Roles	Consolidating users with similar security accesses
Session	Combing global data in a multi-user setting
Full View with Errors	Provide a complete view to users (including exceptions)
Limited View	Displays only data a user can access
Secure Access Layer	Integrates application security with low level security

Table 2.4: Security Pattern Catalog [103]

2.4.2 Non Pattern Based Mitigations

Firesmith *et al.* [35] present a paper that compare and contrast the differences between security and safety requirements. They consider both security and safety requirements to be quality requirements. They define defenseability problem types as:

1. Malicious Harm - Causes Damage to the Asset
2. Security Incidents - Cyber Attacks
3. Security Threats - Perceived Danger
4. Security Risks - A Combination of the Frequency of the Attack and the Resulting Damage

They define defenseability solution types as:

1. Prevention - Malicious harm, Threats or Risks
2. Detection - Malicious harm, Threats or Risks
3. Reaction - Notification and Repair of Malicious harm, Threats or Risks
4. Adaptation - Avoid or Minimize Consequences of Malicious harm, Threats or Risks

They also define setting security requirements, constraints and a taxonomy of security requirements for a system. They also explore the issue with some being opposed to implementing all security countermeasures and finding the compromise to defending a system while not over burdening it with security requirements.

Chapter 3: Approach

3.1 Process

Our goal is to generate a security test suite for an autonomous or semi-autonomous vehicle. This test suite is required to be scaleable and use an MBT approach to test proper mitigation of security attacks. We also want to be able to leverage a behavioral test suite. To keep models smaller we do not want to integrate required security attack mitigations into the primary functional model. Rather, the goal is to keep these two models separate.

Figure 3.1 shows the test generation process. Rectangular boxes show artifacts (models, rules, etc.) while oblong boxes refer to activities. The process consists of four steps. The first generates a behavioral test suite, the second mitigation tests, the third attack scenarios and the forth, the security test suite.

3.1.1 Step 1: Black Box Functional Test Generation

The test generation process starts with a black box model of system behavior, the behavioral model (BM) in Figure 3.1. We associate coverage criteria with BM (i.e. BC) and generate a behavioral test suite (BT). BT can be generated using any graph-based testing criteria from [7] [59] [19] [32] [18] [43] [54].

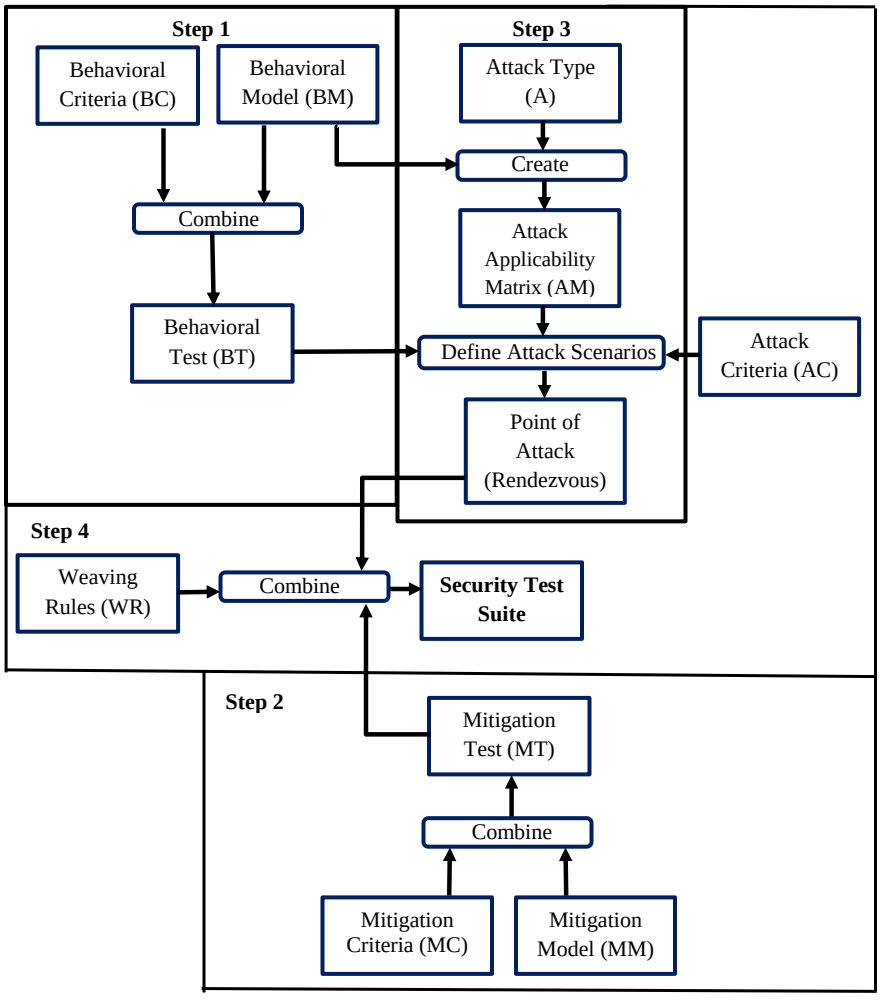

Figure 3.1: Approach

3.1.2 Step 2: Generate Mitigation Tests for all Attack Types

First, we need to determine attack types (A) and determine attack mitigation requirements. These attack mitigation requirements are transformed into attack mitigation models (MM), one for each type of attack. As in the case of the behavioral model, we use coverage criteria (MC) for the mitigation models to direct the generation of test paths (MT) for each mitigation model. Since not all attacks may make sense in every behavioral state, we need to determine in which states an attack may occur. The attack applicability matrix (AM) defines this.

3.1.3 Step 3: Generate Security Test Requirements

Then, we need to determine at which point in the execution of a behavioral test an attack could occur (i.e. where the behavioral and attack process rendezvous and which attack type could occur) These will be our attack security test requirements. We define coverage criteria for them.

3.1.4 Step 4: Generate Security Test Suite

At the point of attack p for attack type a, a required mitigation test path is activated (see Figure 3.2) Weaving rules specify how this mitigation test path is combined with the original test path to create the security test path. The next section describes each step in more detail.

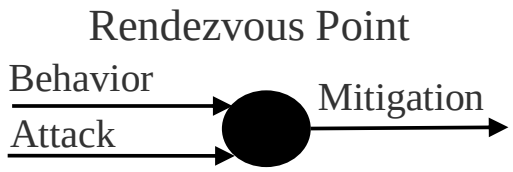

Figure 3.2: Combining Behavioral Test, Attacks and Mitigations

3.2 Step 1: Generating BM with CEFSM

Our approach combines several parallel processes: a behavioral process, an attack process, and a mitigation process. They intersect (communicate) at the point of attack. This is why we need a model capable of representing parallel processes. We chose CEFMs because they have successfully been used to model both distributed systems and network protocols. They can also model orthogonal states of a system in a flat manner and it is not necessary to compose the entire system in one state [37]. CEFSM's consist of a finite set of EFSM's as well as communication channels between the EFSMs [56][37]. CEFSMs are defined as a finite set of consistent and specified EFSMs as well as two disjoint sets of input and output messages [37] [56].

Definition 1: We formally define a CEFSM as follows [37]

CEFSM = (S, s_0, E, P, T, M, V, A, C), such that:

- S is a finite set of states,
- s_0 is the initial state,
- E is a set of events,
- P is a set of boolean predicates,
- T is a transition function such that T: S×P×E→S× A×M,
- M is a set of communicating messages,
- V is a set of variables,
- A is the set of actions, and
- C is the set of communication channels required in this CEFSM.

State changes (action language): The function T returns a next state, a set of output signals, and action list for each combination of a current state, an input signal, and a predicate. The function is defined as:

T(s_i, p_i, get(m_i))/(s_j, A, send(m_{j_1},..., m_{j_k})) where,

- s_i is the current state,
- s_j is the next state,
- p_i is the predicate that must be true in order to execute the transition,
- m_{i_1},..., m_{i_k} are the messages, and

The communicating message m_i is defined as:

(mId, e_j, mDestination) where,

- *mId* is the message identifier, and
- *mDestination* is the CEFSM the message is sent to.

An event e_i is defined as: *(eId, eOccurrence, eStatus)* where,

- *eId* is the event identifier that uniquely identifies it, and
- *eOccurrence* is set to false as long as the event has not occurred for the first time and to true otherwise, and
- *eStatus* is set to true when the event occurs and to false when it no longer applies. Note that *eStatus* allows reoccurring events to happen multiple times. CEFSMs communicate by transferring messages through communication channels C that connect the output of one EFSM to the input of other EFSMs. Let C denote the set $\{\langle name, SYNC|ASYNC\rangle|$ for all the channels in the system$\}$

where *name* is the name of the communication channel and *SYNC* and *ASYNC* indicate that the channel is synchronous or asynchronous. A communication channel can be used differently according to different transitions. A channel $c \in C$ can be represented as $\langle name, t, get()/send() \rangle$ where: [37]

- *name* is the name of the channel
- $t \in T$ refers to the transition linked to this use of the channel
- *get()/send()* indicates whether this channel is an input or an output channel

The action a_i may include an assignment and mathematical operation on the variables. The predicate is defined as a condition that must be met prior to the execution of a function. For example, *T(S_0, e_0, total = 4)/(S_1, {m_0, m_1}, (total = 0; increment(i)))* describes that if a CEFSM is in a state S_0 receives an event e_0 and the value of variable *total* is four at that time, it will move to the next state S_1 and output m_0 and m_1 after setting the *total* to zero and performing *increment(i)* [37].

The formal semantics are defined in [21]. Test criteria such as edge-coverage, prime-path coverage, etc. [7] can be defined. Bourhfir et al. [18] [43] and Hesse [44] propose test generation techniques for CEFSMs. We can use any one of them to create the behavioral test suite BT as a set of paths through BM. Let BT = $\{bt_1..bt_l\}$ be the set of such paths.

3.2.1 Test Case Generation

Once the behavioral model CEFSM has been defined, we can build the behavioral test cases. Behavioral tests are abstract tests and not executable. The example

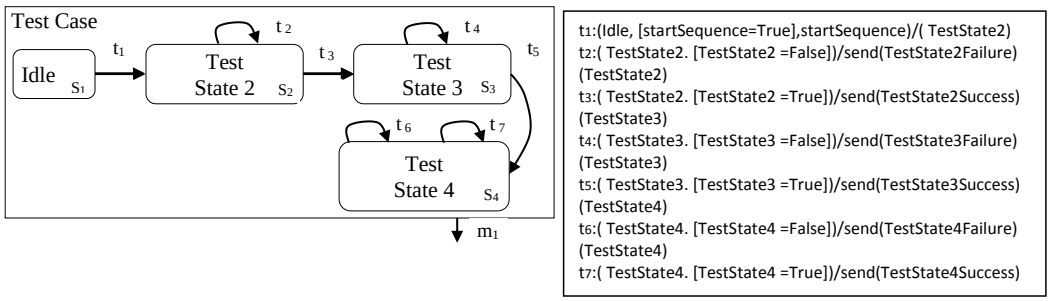

Figure 3.3: Behavioral Test Nomenclature

behavioral test suite nomenclature is illustrated in Figure 3.3. The tests are made up of the current state, completion boolean (to indicate if the test is completed), next state, and message parameters in the following format:

1. Test Path
2. Sequence of nodes
3. Transitions between states
4. Actions
5. Guards
6. Predicates
7. Messages

3.3 Step 2: Identify Attack Types and Mitigations

Software attacks are malicious injections into the system under test (SUT) to attempt to alter or disrupt normal execution. Our attacks are defined in Table 3.1. This dissertation focuses on black box testing, therefore, not all threats described

in Appendix A or threat models in Appendix B are applicable. The attacks we are focused on are:

- Denial of Service (DOS)Radio Frequency (RF) Jam Attack - RF Signal ranges are in the public domain, an attacker can figure out which range of RF signal an autonomous system is transmitting on and send signals in the same range overloading an autonomous vehicle with RF Signals and confusing the system creating a condition called a DOS attack, in which the system is confused and does not know which signals to respond to.

- DOS Flood Attack - An attacker floods an autonomous system with commands or data confusing the system

- Sniff and Spoof Attack - An attacker listens to the data coming from an autonomous system long enough to understand the packet data (for example an attacker might notice that navigation packets are short and video streaming is a larger data packet). Once the packets are understood, the attacker can attempt to Spoof packets similar to the ones it gathered from the vehicle causing an autonomous system to change course or become confused.

- Man in the Middle (MITM) Attack - The attacker gathers data from an autonomous system and passes on either real or modified data to its intended recipient without being recognized.

- GPS JAM Attack - While widely used, GPS has known vulnerabilities in which positioning data can be maliciously altered to cause a vehicle to become confused or crash.

Table 3.1: Summary of Software Attacks

Attack	**Type**	**Description**
a_1	DOS RFJam	Floods RF Channels
a_2	DOS Flood	Floods Comm Channels
a_3	Sniff & Spoof	Listens then injects Packets
a_4	Sniff & MITM	Listens, Forwards and Injects Packets
a_5	DOS GPS JAM	Flood GPS Channels
a_6 ..a_k	Combination	Any attacks can be combined to create a new attack

Definition 2: We formally define a set of attacks as follows

Let a_i be attack type i

for ($i = 1$ *to* k)

Let k be the number of attack types

Let A be the set of attack types

Therefore $A = \{a_1, a_2 \ldots, a_k\}$ represents the set of attack types.

A software attack is an asynchronous process that executes in parallel with functional behavior and interacts with the SUT at particular attack points. Attacks can occur in parallel and in more than one state. We build an attack applicability matrix to identify for each attack whether it can occur in a given state in the behavioral model or not. Attacks can occur as:

1. A Single Attack - For example an attacker may simply sniff the data and do nothing further.

2. Multiple Attacks of Different Types - For example an attacker may attempt to spoof data and create a man in the middle attack.

3. Multiple Attacks of the Same Type - For example an two attackers may attempt to spoof data in the same state.

Definition 3: We formally define an attack applicability matrix as follows:

Let a_j be attack type i

for $(i = 1\ to\ k)$

Let k be the number of attack types

Let A be the set of attack types

Let s_i be the state of the attack

Let i be the index of states in the matrix Let j be the index of attacks in the matrix $(1 \leq a \leq k)$.

$$AM(i,j) = \begin{cases} 1 \text{ if attack } a_j \text{ is applicable in state } s_i \\ 0 \text{ otherwise} \end{cases}$$

$(1 \leq i \leq |S|; 1 \leq j \leq |A|)$

Therefore, AM represents the attack applicability matrix ass a two dimensional array where each column represents a specific behavioral state $s \in S$ and each row is a specific attack type

---

Table 3.2 gives an example of an attack applicability matrix if 5 attacks were possible or if $(k = 5)$. For example, a DOS RF JAM, DOS Flood, and a DOS GPS JAM can happen in any behavioral state, while a sniff and spoof or main in the middle attack only happens in two of the four states. These attacks can only happen after

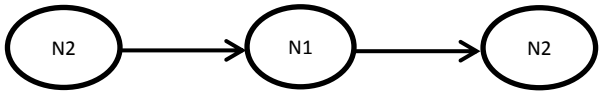

Figure 3.4: M1: Rollback

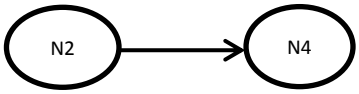

Figure 3.5: M2: Rollforward

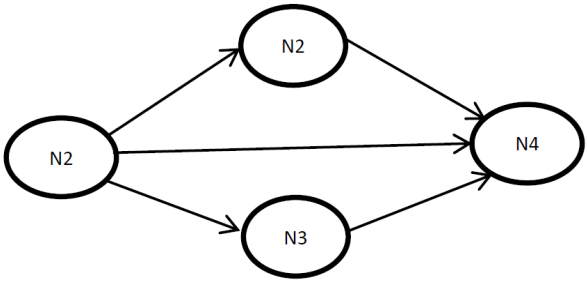

Figure 3.6: M3: Try Other Alternate Mitigation Model

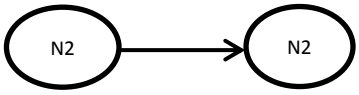

Figure 3.7: M4: Retry

Table 3.2: Attack Applicability Matrix

A/S	s_1	s_2	s_3	s_4
a_1	1	1	1	1
a_2	1	1	1	1
a_3	1	0	1	0
a_4	1	0	0	1
a_5	1	1	1	1

Table 3.3: Software Attacks with Mitigations

Attack	**Type**	**Mitigation**
a_1	DOS RFJam	M1 - Roll Back to Idle State
a_2	DOS Flood	M2 - Roll Forward to Safe State
a_3	Sniff/Spoof	M3 - Encrypt Data
a_4	MITM	M5- Switch to Redundant Services
a_5	GPS JAM	M1- Roll Back to Idle State

the system has moved out of the idle state and into a state where it is transmitting or receiving data. For each attack type, we need to determine how the attack is to be mitigated. A mitigation can consist of many actions. Attack mitigation requirements tend to follow certain patterns. Table 3.3 shows suggested mitigation requirements for the four attack types in Table 3.1. Mitigation requirements now have to be transformed into mitigation models. We define mitigation models as follows[11][58]:

M1: Rollback: return the system to a previous (safe) state in which the attack is not possible. For example, the system may return to an idle state.

M2: Rollforward: jump to the next (safe) state in which the attack is not possible. For example, a launch vehicle may roll forward to an abort state or an autonomous vehicle may roll forward to a "go home" state.

M3: Try other alternatives: pursues an alternative. For example, a network may encrypt data or attempt to use a different frequency.

M4: Retry: the system is rolled back to a previous (safe) state where the attack is not possible, then the activity is re-executed. For example, a system may attempt to retry a command, such as reconnect to the network.

M5: Compensate: the system contains enough redundancy to allow an attack to be masked. For example a launch system generally has a redundant set of avionics hardware or a UAV may have a redundant flight computer. In this case, no mitigation test is required, the behavioral test is executed as is.

Definition 4: Mitigation models are defined as: Let A be the set of attack types Let j be the index of attacks in the matrix $MM_j(1 \leq j \leq |A|)$. Therefore, MM represents the mitigation model for attack type j.

Not all mitigation requirements need a mitigation model. For example, "Compensate" may require the system to take action without any further test input required to identify a problem. Similarly "rollback" shown in Figure Figure 3.4 rolls the system back to a safe state (where the attack is not possible) and re-executes. The mitigation "rollforward", shown in Figure 3.5 rolls the system forward. Figure 3.6 shows an example of a mitigation model for the mitigation pattern "try other alternatives". Lastly, "retry" shown in Figure 3.7, simply retries the command.

Definition 5: Graph-based [7] mitigation criteria is defined as: Let a_i be an attack type

for $(i = 1\ to\ k)$

Let k be the number of attack types

Let A be the set of attack types

Let s_i be the state of the attack

Let i be the index of states in the matrix Let j be the index of attacks in the matrix

Let MC_i be used to generate mitigation test paths

Let $MT_i = mt_{i_1}$, ...,mt_{i_k}

for attack a_i

Therefore, each attack a_i is associated with a mitigation model MM_i

where $i = 1, \ldots, k$.

We can assume that the models are of the same type as the behavioral model BM (i.e. CEFSM). Assuming MC is edge coverage for the mitigation model shown in Figure 3.6. MT consists of two paths:

$MT = \{mt_1..mt_2\}$

$mt_1 = (N_2, N_1, N_2)$

$mt_2 = (N_2, N_4)$

This mitigation would execute a mitigation at the point of attack and then proceed with the remainder of the behavioral test, stop or even go to a safe state. These options are examples of weaving rules. Table 3.3 gives an example of possible mitigation requirements for the four attack types of Table 3.1.

Mitigations vary in complexity. Simple ones such as "Roll Forward" may take the system to an abort state and "Internal Compensate" may switch the system to a redundant avionics system. However, other mitigation models may be more complicated and consist of multiple alternative behaviors.

3.3.1 Combining Software Attacks

This dissertation focuses on basic software attacks. These attacks can be combined in any order to form new attacks. In order to formally combine attacks we look at them as abstract partitions as shown in 3.8. A partition must satisfy:

1. Partitions must cover entire domain space
2. Blocks must be disjoint and not overlap

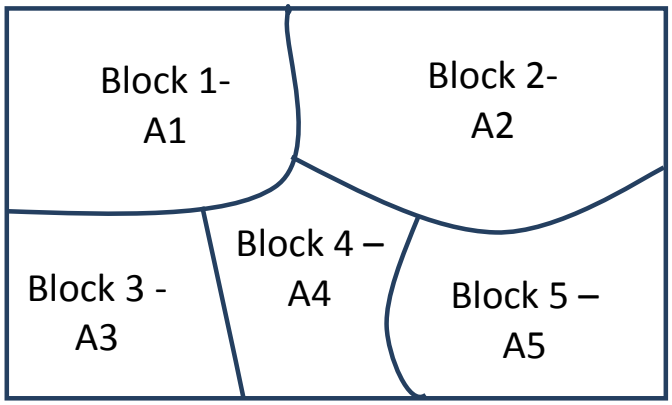

Figure 3.8: Attack Partitions

We can derive coverage criteria from using common combination strategies such as [7]:

1. All Combination Coverage - All combinations used (Would not be practical in systems with more than 2 or 3 blocks)
2. Each Choice Coverage - One value from each block or characteristic used in at least one test case

3. Pair Wise Coverage - A value from each block is combined with a value from every block

4. T-Wise Coverage - A value from each block for each group of T characteristic must be combined. T should be selected between 2 and the number of partitions. In our case, it would be between 2 and 5.

5. Base Choice Coverage - A base test is selected and all other tests are chosen from non-base tests.

6. Multiple Base Choice Coverage - Multiple base tests are selected and all other tests are chosen from non-base tests.

3.4 Step 3: Generate Security Test Requirements

The rendezvous point (attack point) is the intersection where the intended behavior and a single attack or multiple attacks occur. Some attacks have preconditions and others can occur in parallel. Security test requirements specify where in the behavioral test $bt_i \in BT$ the attack is to occur (i.e. figure 3.2) and in which $bt_i \in BT$ the attack should occur. The top rows of Tables 3.4, 3.5, 3.6 and 3.7 show an example of CT. We define four different coverage criteria. For simplicity, and to express the behavioral test suite as attack points along a fixed length.

Definition 6: We formally define a security test requirement as follows:

Let CT be the concatenation of test paths in BT.

That is $CT = \{bt_1 \circ bt_2 \circ bt_3 \ldots bt_l\}$

Let len(bt) be the number of nodes in bt.

Let $I = len(CT) = \sum_{i=1}^{l} len(bt_i)$

Let the rendezvous point p is a position in the behavioral test suite where a single or multiple attacks can occur ($1 \leq p \leq I$)

Let an attack type a ($1 \leq a \leq |\mathrm{A}|$) be applied at position p

Let the attack scenario be defined as (p,a) The set of attack scenarios constitutes the security test requirements for which security tests need to be generated.

Let a pair (p, a) define an attack scenario in terms of where in CT an attack is to be injected.

Assume: we can only select a feasible attack scenario.

Let node (p) be the node s in position p.

Then an attack scenario (p, a) is feasible if and only if AM(node(p), a))= 1.

We use coverage criteria to determine a set of attack scenarios (i.e. security test requirements) as follows:

Let s be a state in the behavioral model.

Assume: there is at least one state in any behavioral model where an attack can be injected.

Therefore, for a given attack type $a \in A$ there must be some node s such that $AM(s, a) = 1$ is true.

Attack coverage criteria AC for selecting (s, a) must be defined.

Example: This example demonstrates the differences between the four types of coverage criteria for all combinations (p, a). Suppose we have a test suite that has two test paths, i.e. $CT = \{bt_1\}$ where

$bt_1 = \{s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_4\}$, then

I = [1,10]. Assume we have 5 attack types and the attack applicability matrix of Table 3.2.

A = $\{a_1, a_2, a_3, a_4, a_5\}$; |A| = 5.

Criteria 1: All feasible combinations of (p, a). Criteria 1 would require to select all attack scenarios(p, a) where AM(node(p), a)= 1. Table 3.4 specifies 31 such security test requirements based on AM of Table 3.2.

Criteria 2: All unique nodes, all applicable attacks. This only requires $\sum_{j=1}^{k}\sum_{i=1}^{5}$

Table 3.4: Position Attack Matrix- Criteria 1

	bt_1							
A/CT	s_1	s_2	s_2	s_3	s_3	s_4	s_4	s_4
$\boldsymbol{a}_1$	1	1	1	1	1	1	1	1
$\boldsymbol{a}_2$	1	1	1	1	1	1	1	1
$\boldsymbol{a}_3$	1	0	0	1	1	0	0	0
$\boldsymbol{a}_4$	1	0	0	0	0	1	1	1
$\boldsymbol{a}_5$	1	1	1	1	1	1	1	1

(AM(i,j)=1) combinations, i.e. the number of "1" entries in the applicability matrix. When nodes occur multiple times in a test suite only one needs to be selected. This could lead to not testing attack mitigation in all tests $bt_i \in BT$. Table 3.5 shows an example of (p, a) attack scenarios that fulfill this criteria. There are 16 such security test requirements.

Table 3.5: Position Attack Matrix- Criteria 2

	bt_1							
A/CT	s_1	s_2	s_2	s_3	s_3	s_4	s_4	s_4
$\boldsymbol{a}_1$	1	1	0	1	0	1	0	0
$\boldsymbol{a}_2$	1	1	0	1	0	1	0	0
$\boldsymbol{a}_3$	1	0	0	1	0	0	0	0
$\boldsymbol{a}_4$	1	0	0	0	0	1	0	0
$\boldsymbol{a}_5$	1	1	0	1	0	1	0	0

Criteria 3: All tests, all unique nodes, all attacks which are applicable. In addition to criteria 2, this criterion requires to select positions representing all tests $bt_i \in$ BT. Table 3.6 shows a set of attack requirements that meets this criterion. Note that we end up with the same number of security test requirements as in criteria 2 but different positions (in different tests).Note that criteria 3 subsumes criteria 2. As before, there are 16 security test requirements.

Table 3.6: Position Attack Matrix- Criteria 3

	bt_1							
A/CT	s_1	s_2	s_2	s_3	s_3	s_4	s_4	s_4
$\boldsymbol{a}_1$	1	0	1	0	1	0	0	1
$\boldsymbol{a}_2$	1	0	1	0	1	0	0	1
$\boldsymbol{a}_3$	1	0	0	0	1	0	0	0
$\boldsymbol{a}_4$	1	0	0	0	0	0	0	1
$\boldsymbol{a}_5$	1	0	1	0	1	0	0	1

Criteria 4: All tests, all unique states, some attacks. All attacks must be paired with a state at least once, but not with every state. Table 3.7 shows a set of security test requirements that meets this criteria. We only have 5 security test requirements.

Table 3.7: Position Attack Matrix- Criteria 4

	bt_1							
A/CT	s_1	s_2	s_2	s_3	s_3	s_4	s_4	s_4
$\boldsymbol{a}_1$	1	0	0	0	0	0	0	0
$\boldsymbol{a}_2$	0	1	0	0	0	0	0	0
$\boldsymbol{a}_3$	0	0	0	1	0	0	0	0
$\boldsymbol{a}_4$	0	0	0	0	0	1	0	0
$\boldsymbol{a}_5$	0	1	0	0	0	0	0	0

3.4.1 Multiple Attacks

The four criteria described above cover a single attack. Multiple attacks are also possible. We define two types of multiple attacks. The first being, multiple attacks of different types. An attacker may attempt two entirely different attacks in the same state. For example, an attacker may attempt to spoof data and create a man in the middle attack at the same time. Our approach would cover this attack as a new attack type.

We also define multiple attacks of the same type. An attacker may attempt to execute the same attack twice in the same states. For example, An attacker may attempt to spoof data in 2 separate attacks in the same state. Our approach would consider this attack simply as a new attack type.

3.4.2 Attack Priorities

We also need to set priorities for some attack combinations and states. Attacks that would cause the most harm to the system are given a higher priority. Attacks such

Table 3.8: Software Attack Priorities

Attack	**Type**	**Priority**
a_1	DOS RFJam	2
a_2	DOS Flood	2
a_3	Sniff & Spoof	1
a_4	Sniff & MITM	1
a_5	DOS GPS	2

as Sniff and Spoof and Sniff and Man in the middle are given the highest priority as they have the potential of manipulating the system's behavior by injecting malicious commands. Attacks such as DOS RF JAM, DOS FLOOD and DOS GPS JAM are also very dangerous, but they are at a lower level, because they should be the easiest to identify and mitigate. Let SR_i be the set of (p, a) pairs selected by criteria $C_i(i - 1...4)$. SR_i forms our set of security test requirements.

3.5 Step 4: Security Test Generation

We generate a security mitigation test (SMT). The SMT is comprised from data gathered in the mitigation pattern table shown in Table 3.9. Test paths are then then generated by examining the test cases defined in criteria 4 and weaving in the proper mitigation. The SMT is definition as:

Definition 7: Let CT be the concatenated test path

Let a be an attack type

Let p be a position in CT

Let SR be a security test requirement

Let i be an index in SR

For each requirement $(p, a) \in SR_i$,

Let $bt \in BT$ be the test which includes position $p \in CT$

Let $mt_a \in MT_a$ be a mitigation for attack type a

Let $s = node(p)$ be the behavioral state in position p of CT. Therefore, we create $smt \in SMT$ with the weaving rules $wr_a \in WR_a$ as follows:

1. Keep path represented by bt until state s
2. Apply attack a at state s
3. Apply weaving rule wr_a to construct smt

Our weaving rules are similar to [8]. We formally define them for each type of attack.

Let t = { s_1...*node(p)*...s_k }

WR1: Fix

1. Option 1 - Compensate: (Partial Fix and proceed) mitigates the attack and continues with the remainder of the behavioral test. So, $smt = s_1 \ldots node(p) mt_a \; node(p) \ldots s_k$. mt_a may be zero, if mitigation does not require a user input.
2. Option 2 -Fix and stop: Mitigates an attack and ignores the remainder of $t : smt = s_1 \ldots node(p) mt_a$.

WR2: Rollforward

1. Option 1 - Rollforward: mitigates the attack, and proceeds. Therefore, $smt =$ s_1...*node(n)* $mt_a s_f \ldots s_k$ where s_f is the node in bt to which we rollforward. If no other actions are required then mt_a is empty and $smt = s_1 \ldots node(p) s_f \ldots s_k$.

2. Option 2 - Deferred Fixing: If the failure can only be fixed after reaching the rollforward node s_f then smt becomes:
   $smt = s_1 \ldots node(p) s_f mt_a s_{f+1} \ldots s_k$. Note that further variations of this weaving rule can exist, like a state s_d between s_f and s_k at which the failure mitigation mt_a is inserted.
   $bt = s_1 \ldots s_b \ldots node(p) \ldots s_f \ldots s_d \ldots s_k$
   $smt = s_1 \ldots node(p) s_f \ldots s_d mt_a \ldots s_k$

WR3: Rollback

1. Option 1 - Rollback: Apply mitigation path mt_a from point of attack to state in which attack is not applicable and continue with remainder of behavioral test. $smt = s_1 \ldots node(p) mt_a s_b \ldots s_k$ where s_b is a node before node(p).
2. Option 2 - Rollback and Stop: $smt = s_1 \ldots node(p) mt_a s_b$
3. Option 3 - Retry Once: $smt = s_1 \ldots node(p) node(p)^r \ldots s_k$ where $r = 1$.

WR4: Internal compensate (no action required)
This attack is mitigated when a system switches to a redundant set of hardware. The test would include applying the attack and continuing to execute the test bt. As attacks evolve, mitigations and weaving rules may change as well.

We now proceed to generate an attack mitigation for each $(p, a) \in SR_i$ The results of applying these weaving rules is SMT. We have to select input values to make the tests executable. We define an example security test suite as shown in Table 3.10. This

Table 3.9: Example Mitigation Pattern

State	Attack	Risk	MT Mitigation	WR
s_1	a_1		Roll Forward to Abort	WR2 Option 1
	a_2		Roll Forward to Abort	WR2 Option 1
	a_3		Roll Forward to Abort	WR2 Option 1
	a_4		Roll Forward to Abort	WR2 Option 1
	a_5		Roll Forward to Abort	WR2 Option 1
s_2	a_1		Roll Back to s1	WR3 Option 1
	a_2		Roll Back to s1	WR3 Option 1
	a_3	NA		
	a_4	NA		
	a_5		Roll Back to s1	WR3 Option 1
s_3	a_1		Roll Back to s1	WR3 Option 1
	a_2		Roll Back to s1	WR3 Option 1
	a_3		Roll Back to s1	WR3 Option 1
	a_4	NA		
	a_5		Roll Back to s1	WR3 Option 1
s_4	a_1		Roll Back to s1	WR3 Option 1
	a_2		Roll Back to s1	WR3 Option 1
	a_3	NA		
	a_4		Roll Back to s1	WR3 Option 1
	a_5		Roll Back to s1	WR3 Option 1

test suite assumes roll back mitigation and covers the tests that would be required for criteria 4 noted above.

Table 3.10: Security Mitigation Test for Criteria 4

Security Test	**SMT**	**bt**
smt_1	s_1, N_2, N_4	bt_1
smt_2	$s_1, s_2, N_2, N_1, N_2, s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_4$	bt_1
smt_3	$s_1, s_2, s_2, s_3, N_2, N_1, N_2, s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_4$	bt_1
smt_4	$s_1, s_2, s_2, s_3, s_3, s_4, N_2, N_1, N_2, s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_4$	bt_1
smt_5	$s_1, s_2, N_2, N_1, N_2, s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_4$	bt_1

We complete the example by demonstrating executable tests. Our tests have been built using the SMT shown in Table 3.10 by adding variables and messages. Our executable tests demonstrate the weaving rules and mitigations interlaced into the test paths.

$SMT1:$

$s_1 : (Idle, [startSequence = True],$

$startSequence/send(TestState1Success)(TestState2)$

$s_2 : (TestState2, [TestState2 = True],$

$TestState2Success/send(TestState2Success)(TestState2)$

$N_2 : (Attack, [Attack = True][AttackType = a_1]$

$RolllForward/send(AbortMessage)(RollForwardAbort)$

$N_4 : (RollForwardAbort, [Abort = True],$

$AbortSuccess/send(AbortSuccess)(EndSequence);$

$SMT2:$

$s_1 : (Idle, [startSequence = True],$

$startSequence/send(TestState1Success)(TestState2)$

$s_2 : (TestState2, [TestState2 = True],$

$TestState2Success/send(TestState2Success)(TestState2)$

$N_2 : (Attack, [Attack = True][AttackType = a_2]$

$RollBack/send(RollbackMessage)(RollBackSafe)$

$N_1 : (RollBackSafe, [SafeState = s_1],$

$RollBackSuccess/send(RollBackSuccess)(RollBackSafe)$

$N_2 : (Attack, [Attack = True][AttackType = a_2]$

$RollBack/send(RollbackMessage)(RollBackSafe)$

$s_1 : (Idle, [startSequence = True],$

$startSequence/send(TestState1Success)(TestState2)$

$s_2 : (TestState2, [TestState2 = True],$

$TestState2Success/send(TestState2Success)(TestState2)$

$s_2 : (TestState2, [TestState2 = True],$

$TestState2Success/send(TestState2Success)(TestState3)$

$s_3 : (TestState3, [TestState3 = True], TestState3Success$

$/send(TestState3Success)(TestState3)$

$s_3 : (TestState3, [TestState3 = True],$

$TestState3Success/send(TestState3Success)(TestState4)$

$s_4 : (TestState4, [TestState4 = True],$

$TestState4Success/send(TestState4Success)(TestState4)$

$s_4 : (TestState4, [TestState4 = True],$

$TestState4Success/send(TestState4Success)(TestState4)$

$s_4 : (TestState4, [TestState4 = True],$

$TestState4Success/send(TestState4Success)(EndSequence);$

$SMT3 :$

$s_1 : (Idle, [startSequence = True],$

$startSequence/send(TestState1Success)(TestState2)$

$s_2 : (TestState2, [TestState2 = True],$

$TestState2Success/send(TestState2Success)(TestState2)$

$s_2 : (TestState2, [TestState2 = True],$

$TestState2Success/send(TestState2Success)(TestState3)$

$s_3 : (TestState3, [TestState3 = True],$

$TestState3Success/send(TestState3Success)(TestState3)$

$N_2 : (Attack, [Attack = True][AttackType = a_3]$

$RollBack/send(RollbackMessage)(RollBackSafe)$

$N_1 : (RollBackSafe, [SafeState = s_1],$

$RollBackSuccess/send(RollBackSuccess)(RollBackSafe)$

$N_2 : (Attack, [Attack = True][AttackType = a_3]$

$RollBack/send(RollbackMessage)(RollBackSafe)$

$s_1 : (Idle, [startSequence = True],$

$startSequence/send(TestState1Success)(TestState2)$

$s_2 : (TestState2, [TestState2 = True],$

$TestState2Success/send(TestState2Success)(TestState2)$

$s_2 : (TestState2, [TestState2 = True],$

$TestState2Success/send(TestState2Success)(TestState3)$

$s_3 : (TestState3, [TestState3 = True], TestState3Success$

$/send(TestState3Success)(TestState3)$

$s_3 : (TestState3, [TestState3 = True],$

$TestState3Success/send(TestState3Success)(TestState4)$

$s_4 : (TestState4, [TestState4 = True],$

$TestState4Success/send(TestState4Success)(TestState4)$

$s_4 : (TestState4, [TestState4 = True],$

$TestState4Success/send(TestState4Success)(TestState4)$

$s_4 : (TestState4, [TestState4 = True],$

$TestState4Success/send(TestState4Success)(EndSequence);$

$SMT4 :$

$s_1 : (Idle, [startSequence = True],$

$startSequence/send(TestState1Success)(TestState2)$

$s_2 : (TestState2, [TestState2 = True],$

$TestState2Success/send(TestState2Success)(TestState2)$

$s_2 : (TestState2, [TestState2 = True],$

$TestState2Success/send(TestState2Success)(TestState3)$

$s_3 : (TestState3, [TestState3 = True],$

$TestState3Success/send(TestState3Success)(TestState3)$

$s_3 : (TestState3, [TestState3 = True],$

$TestState3Success/send(TestState3Success)(TestState4)$

$s_4 : (TestState4, [TestState4 = True],$

$TestState4Success/send(TestState4Success)(TestState4)$

$N_2 : (Attack, [Attack = True][AttackType = a_4]$

$RollBack/send(RollbackMessage)(RollBackSafe)$

$N_1 : (RollBackSafe, [SafeState = s_1],$

$RollBackSuccess/send(RollBackSuccess)(RollBackSafe)$

$N_2 : (Attack, [Attack = True][AttackType = a_4]$

$RollBack/send(RollbackMessage)(RollBackSafe)$

$s_1 : (Idle, [startSequence = True],$

$startSequence/send(TestState1Success)(TestState2)$

$s_2 : (TestState2, [TestState2 = True],$

$TestState2Success/send(TestState2Success)(TestState2)$

$s_2 : (TestState2, [TestState2 = True],$

$TestState2Success/send(TestState2Success)(TestState3)$

$s_3 : (TestState3, [TestState3 = True], TestState3Success$

$/send(TestState3Success)(TestState3)$

$s_3 : (TestState3, [TestState3 = True],$

$TestState3Success/send(TestState3Success)(TestState4)$

$s_4 : (TestState4, [TestState4 = True],$

$TestState4Success/send(TestState4Success)(TestState4)$

$s_4 : (TestState4, [TestState4 = True],$

$TestState4Success/send(TestState4Success)(TestState4)$

$s_4 : (TestState4, [TestState4 = True],$

$TestState4Success/send(TestState4Success)(EndSequence);$

$SMT5 :$

$s_1 : (Idle, [startSequence = True],$

$startSequence/send(TestState1Success)(TestState2)$

$s_2 : (TestState2, [TestState2 = True],$

$TestState2Success/send(TestState2Success)(TestState2)$

$N_2 : (Attack, [Attack = True][AttackType = a_5]$

$RollBack/send(RollbackMessage)(RollBackSafe)$

$N_1 : (RollBackSafe, [SafeState = s_1],$

$RollBackSuccess/send(RollBackSuccess)(RollBackSafe)$

$N_2 : (Attack, [Attack = True][AttackType = a_5]$

$RollBack/send(RollbackMessage)(RollBackSafe)$

$s_1 : (Idle, [startSequence = True],$

$startSequence/send(TestState1Success)(TestState2)$

$s_2 : (TestState2, [TestState2 = True],$

$TestState2Success/send(TestState2Success)(TestState2)$

$s_2 : (TestState2, [TestState2 = True],$

$TestState2Success/send(TestState2Success)(TestState3)$

$s_3 : (TestState3, [TestState3 = True], TestState3Success$

$/send(TestState3Success)(TestState3)$

$s_3 : (TestState3, [TestState3 = True],$

$TestState3Success/send(TestState3Success)(TestState4)$

$s_4 : (TestState4, [TestState4 = True],$

$TestState4Success/send(TestState4Success)(TestState4)$

$s_4 : (TestState4, [TestState4 = True],$

$TestState4Success/send(TestState4Success)(TestState4)$

$s_4 : (TestState4, [TestState4 = True],$

$TestState4Success/send(TestState4Success)(EndSequence)$;

3.6 Integrating New Attacks

We will add a new attack A_6 to the attack matrix in Table 3.11. We will also update the attack applicability matrix by adding a row in Table 3.12 to give example states in which our new attack is possible. Suppose we have a test suite with two test paths, i.e. $CT = \{bt_1\}$ where $bt_1 = \{s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_4\}$,
then I = [1,10]. Assume we have 6 applicable single attacks
A = $\{a_1, a_2, a_3, a_4, a_5, a_6\}$; |A|.
As shown in Table 3.13, Table 3.14, Table 3.15 and Table 3.15. The only tests that would need to be performed would be in the row labeled A_r as they are the only tests that have been modified.

Table 3.11: Existing Software Attacks with New Attack to Test

Attack	**Type**	**Description**
a_1	DOS RFJam	Floods RF Channels
a_2	DOS Flood	Floods Comm Channels
a_3	Sniff & Spoof	Listens then injects Packets
a_4	Sniff & MITM	Listens, Forwards and Injects Packets
a_5	DOS GPS JAM	Flood GPS Channels
a_6	New Attack	

Table 3.12: Attack Applicability Matrix

A/S	$\boldsymbol{s}_1$	$\boldsymbol{s}_2$	$\boldsymbol{s}_3$	$\boldsymbol{s}_4$
$\boldsymbol{a}_6$	1	1	1	1

Table 3.13: New Attack Position Matrix- Criteria 1

	bt_1							
A/CT	s_1	s_2	s_2	s_3	s_3	s_4	s_4	s_4
$\boldsymbol{a}_6$	1	1	1	1	1	1	1	1

Table 3.14: New Attack Position Matrix- Criteria 2 and 3

	bt_1							
A/CT	s_1	s_2	s_2	s_3	s_3	s_4	s_4	s_4
$\boldsymbol{a}_6$	1	1	0	1	0	1	0	0

Table 3.15: New Attack Position Matrix- Criteria 4

	bt_1							
A/CT	s_1	s_2	s_2	s_3	s_3	s_4	s_4	s_4
$\boldsymbol{a}_6$	0	0	0	0	0	1	0	0

Chapter 4: Case Studies

4.1 Case Study Research Questions

We demonstrate our approach using four case studies. Our case studies include a launch vehicle (both pre and post launch), a UAV and a robot. We will use the case studies to answer the following questions:

- CSQ1: Can we show applicability to various systems?

- CSQ2: Can we demonstrate generalization in different domains?

- CSQ3: Can we prove effectiveness in detecting and mitigating attacks?

- CSQ4: Can we exhibit efficiency in the size of our test suites?

4.2 Case Study 1: Pre-Launch System

In this section we demonstrate our approach with a launch vehicle example to show how to test for security attacks. The goal of this case study is to demonstrate our approach on a critical system. Launch vehicles such as the Atlas 5 or Delta 4 are used to deploy satellites into space. Launch vehicles have also been used in silos to deploy war heads, such as the Minuteman. The potential harm in a launch vehicle being compromised is huge. On the launch pad they could explode which could result in loss of life and loss of the vehicle.

A launch system consists of a launch conductor, ground system, launch pad, mobile launch platform and an expendable launch vehicle which is comprised of a booster, upper stage and a payload. Launch vehicles are classified by the number of stages they contain. A vehicle can contain between one and five stages. For this example the vehicle will have two stages. Launch vehicles are also characterized by how much weight they can carry into orbit. Launch vehicles are capable of delivering payloads to low earth orbit (LEO), medium earth orbit (MEO) and geostationary earth orbit (GEO).

- small lift - Capable of Lofting 4400 lbs into LEO

- medium lift - Capable of Lofting up to 44,100 lbs into LEO

- heavy lift - Capable of Lofting up to 110,000 lbs into LEO

The launch pad can be a spaceport, fixed missile silo, or mobile seaport. In this example the launch pad will be a spaceport. The booster and upper stage are fueled by cryogenic fuels which can only be liquefied at extremely low temperatures. Cryogenic fuels, such as liquid hydrogen (LH2) and liquid oxygen (LOX) are selected because they are widely available and inexpensive. They are also chosen because they generate a high specific impulse(up to 4.4 km/s), which defines their efficiency of fuel relative to the amount consumed.

The launch controller is responsible for initiating the launch sequence and verifying the safety and security of the launch control system throughout the launch. The launch conductor communicates to the vehicle through the ground system. The ground system is physically connected to the launch vehicle via Ethernet cables, serial cables, 1553 data cables and fuel lines.

The sequence begins twenty four hours before a launch. The launch conductor initiates a network connection. This action powers on hazard lights (include both

search lights and amber warning lights) to indicate to launch personnel that the launch pad is "hot" (or a launch is imminent). The launch conductor clears the area of non-essential personnel using a public announcement system and begins the countdown clock.

The launch conductor then initiates an environmental control system (ECS) on the launch pad. This includes soliciting a weather briefing, performing an air conditioning check as well as a nitrogen purge function. When the ECS is complete, the system transitions into a fuel check mode.

The mobile launch platform (MLP) and vehicle are moved to the spaceport launch pad. A cryogenic fuel check is completed by verifying the launch vehicle's Liquid Oxygen LO2, helium and Liquid Hydrogen LH2. These fuels are chosen because as noted above, they are the most efficient launch vehicle fuels. They are also highly explosive and must be checked often.

The system moves into a pre-flight stage, where an instrumentation check is performed and cryogenic testing is completed. The launch conductor initiates a chill down procedure (which is used to keep the cryogenic fuels at the proper temperature), then a battery check is performed. The launch conductor verifies fuel pressures and initiates fueling if the pressures are low. The launch vehicle system is now ready for flight.

The launch conductor prepares the vehicle by switching the launch system to run on its own internal battery power and begins sending the flight command to initiate the launch. If all is successful, the vehicle successfully lifts off of the launch pad. Should there be an issue after launch, the flight can be terminated using a safe arm device (SAD).

Figures 4.1 and 4.2 show the CEFSM model of the launch system including states, transitions, variables, events, and messages. The CEFSM begins with the initializa-

tion phase. In this phase, the model transitions with t_1 from an "idle" state s_1 (where no attacks are possible) to the "network connection" state s_2. If the "network connection" s_2 state fails, retry's are attempted with t_2. if the "network connection" state s_2 is successful, a success message is sent and the model transitions with t_3 to the "hazards lights on" state s_3. The "hazards lights on" state s_3 is retried with a t_2 transition on a failure. For a success, the "hazards lights on" state s_3 sends a success message and transitions to a "countdown clock reset" state s_4. The model transitions to environmental system control (ESC) "initialization state" s_5, which is considered to be a safe state and no attacks are possible. The model then transitions with t_8 to the "air conditioning" state s_6, which performs an air conditioning check of the vehicle. If this is successful, the model transitions using t_{10} to the "nitrogen purge" state s_7. This completes the ESC initialization and the model transitions with t_{12} to s_8 Fuel check "initialization state". This state s_8 is also considered a safe state with no attacks possible. The model transitions with t_{13} to begin a cryogenic fuel check is completed by verifying the launch vehicle's Liquid Oxygen LO2 in state s_9, transitions with t_{15} to a "helium state" s_{10} and transitions with t_{17} to the "Liquid Hydrogen LH2" state s_{11}. When the fuel check is complete, the model transitions with t_{19} to a pre-flight stage. The "initialization state" s_{12} is also a state state. The model transitions using t_{20}, where an "instrumentation check" is performed in s_{13} transitions with t_{22} to a "cryogenic testing" state s_{14}. The model transitions with a t_{24} to the "chill down" state s_{15}, (which is used to keep the cryogenic fuels at the proper temperature). Then the model transitions with t_{26} to a "battery check" state s_{16}. Then, the model transitions with t_{28} to the "initiate fueling" state s_{17} which verifies fuel pressures and initiates fueling if the pressures are low. The launch vehicle system is now ready for flight and transitions with t_{30} to a fight "initialize state" s_{18} (considered a safe state). The vehicle transitions with t_{31} to the "internal battery"

state s_{19}, which prepares the vehicle by switching the launch system to run on its own internal battery power. The model then transitions with t_{33} to the "flight command" state s_{20}, so that the vehicle can accept the flight command to initiate the launch.

4.2.1 Security Attacks on a Launch System

Security attacks can be injected at each state where there is a communication with the ground system. Attacks on the initialization stages are less extreme, as the fueling has not yet begun and the issue may be resolved so that the launch can continue. The initialization sequence includes network connection, countdown clock and hazard lights states. Any of these can be mitigated with a retry before an abort command is issued. The remaining states are the the most critical and could result in explosion of the launch system. If these states have experienced an attack, the only mitigation is to abort or initiate the SAD. The attacks that are possible in a pre-flight systems are defined in Table 4.1.

Table 4.1: Possible Pre-flight Launch Vehicle Attacks

Attack	**Type**	**Description**
a_1	DOS RFJam	Floods RF Channels
a_2	DOS Flood	Floods Comm Channels
a_3	Sniff & Spoof	Listens then injects Packets
a_4	Sniff & MITM	Listens, Forwards and Injects Packets

4.2.2 Launch Vehicle Behavioral Model

Figure 4.1 depicts the behavioral model of a launch vehicle in Communicating Extended Finite State Machine (CEFSM) format. The model contains 21 unique states and 34 transitions. Test paths are determined by following both states and transi-

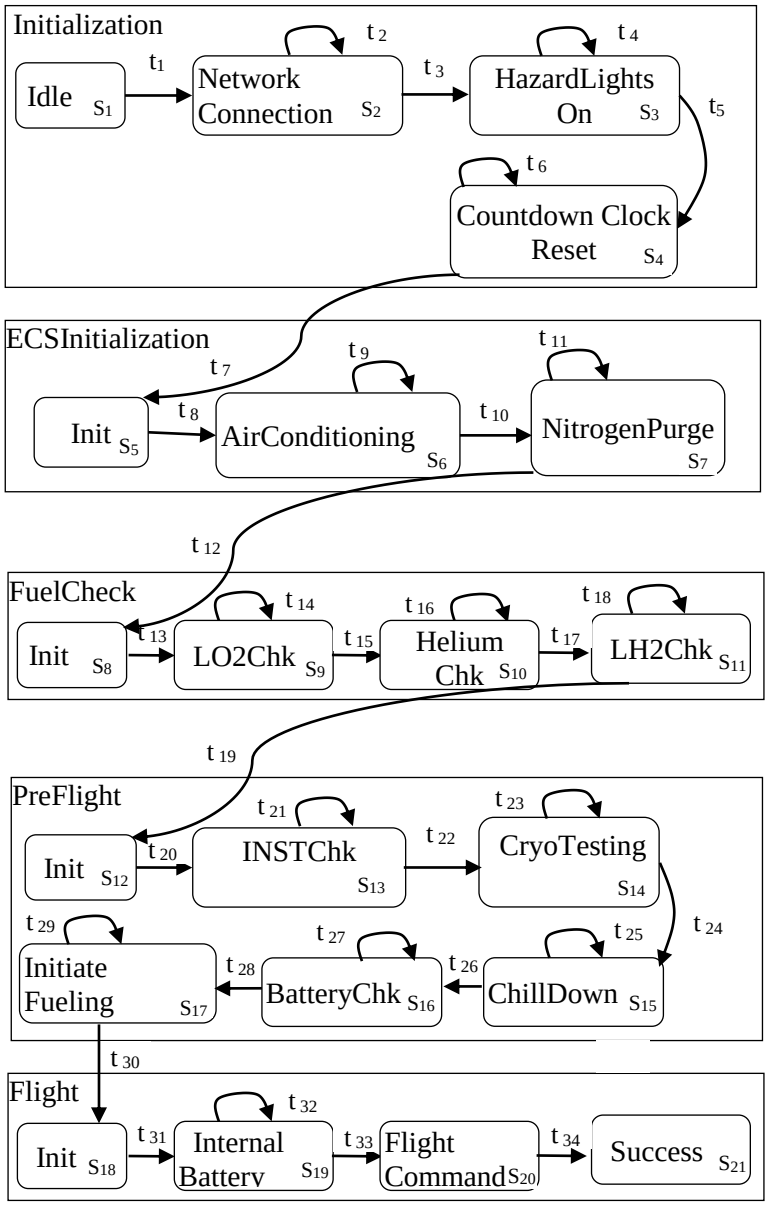

Figure 4.1: Pre-Flight Launch Vehicle CEFSM Part 1

t1:(Idle, [startSequence=True],startInitialize)/(NetworkConnection)

t2:(NetworkConnection. [NetworkConnection =False])/send(NetworkConnectionFailure)(NetworkConnection)

t3:(NetworkConnection. [NetworkConnection =True])/send(NetworkConnectionSuccess) (HazardLightsOn,)

t4:(HazardLightsOn. [HazardLightsOn =False])/send(HazardLightsOn Failure) (HazardLightsOn,)

t5:(HazardLightsOn. [HazardLightsOn =True])/send(HazardLightsOn Success) (CountdownClockReset)

t6:(CountdownClockReset. [CountdownClockReset =False])/send(CountdownClockResetFailure) (CountdownClockReset)

t7:(CountdownClockReset. [CountdownClockReset =True]) /send(CountdownClockResetSuccess)(Init)

t8:(Init. [startSequence=True],start ECSInitialization)/ (AirConditioning)

t9:(AirConditioning. [AirConditioning =False)/send(AirConditioningFailure) (AirConditioning)

t10:(AirConditioning. [AirConditioning =True])/send(AirConditioning Success)(NitrogenPurge)

t11:(NitrogenPurge. [NitrogenPurge =False])/send(NitrogenPurgeFailure) (NitrogenPurge)

t12:(NitrogenPurge. [NitrogenPurge =True])/send(NitrogenPurgeSuccess) (Init)

t13: (Init. [startSequence=True],LO2Chk)/ (LO2Chk)

t14:(LO2Chk. [LO2Chk =False])/send(LO2ChkFailure)(LO2Chk)

t15:(LO2Chk. [LO2Chk =True])/send(LO2ChkSuccess) (HeliumChk)

t16:(HeliumChk. [HeliumChk =False])/send(HeliumChkFailure)(HeliumChk)

t17:(HeliumChk. [HeliumChk =True]])/send(HeliumChkSuccess)(LH2Chk)

t18:(LH2Chk. [LH2Chk =False])/send(LH2ChkFailure)(LH2Chk)

t19:(LH2Chk [LH2Chk =True]])/send(LH2ChkSuccess) (Init)

t20: (Init. [startSequence=True],startPreflight)/ (INSTChk)

t21:(INSTChk. [INSTChk =False])/send(INSTChkFailure) (INSTChk)

t22:(INSTChk. [INSTChk =True])/send((INSTChkSuccess)(CryoTesting)

t23:(CryoTesting. [CryoTesting =False])/send(CryoTestingFailure)(CryoTesting)

t24:(CryoTesting. [CryoTesting =True])/send(CryoTestingSuccess)(ChillDown)

t25:(ChillDown. [ChillDown =False])/send(ChillDownFailure)(ChillDown)

t26:(ChillDown. [ChillDown =True])/send(ChillDownSuccess) (BatteryCheck)

t27:(BatteryCheck. [BatteryCheck =False])/send(BatteryCheckFailure)(BatteryCheck)

t28:(BatteryCheck. [BatteryCheck =True])/send(BatteryCheckSuccess) (InitiateFueling)

t29:(InitiateFueling. [InitiateFueling =False])/send(InitiateFuelingFailure) (InitiateFueling)

t30:(InitiateFueling. [InitiateFueling =True])/send(InitiateFuelingSuccess) (Init)

t31: (Init. [startSequence=True],Flight)/ (InternalBattery)

t32:(InternalBattery. [InternalBattery =False])/send(InternalBatteryFailure) (InternalBattery)

t33:(InternalBattery. [InternalBattery =True])/send(InternalBatterySuccess) (FlightCommand)

t34:(FlightCommand. [FlightCommand =True])/send(Success)

Figure 4.2: Pre-Flight Launch Vehicle CEFSM Part 2

Table 4.2: Pre-flight Launch Attacks

State	Attack	Risk	MT Mitigation	WR
s_1		NA		
s_2	a_1	No Network	M1/M4 Rollback to s1	WR3 Option 1
	a_2	No Network	M1/M4 Rollback to s1	WR3 Option 1
s_3	a_1	Confusion	M1/M4 Rollback to s1	WR3 Option 1
	a_2	Confusion	M1/M4 Rollback to s1	WR3 Option 1
	a_3	Loss of Control	M1/M4 Rollback to s1	WR3 Option 1
	a_4	Loss of Control	M1/M4 Rollback to s1	WR3 Option 1
s_4	a_1	Confusion	M1/M4 Rollback to s1	WR3 Option 1
	a_2	Confusion	M1/M4 Rollback to s1	WR3 Option 1
	a_3	Loss of Control	M1/M4 Rollback to s1	WR3 Option 1
	a_4	Loss of Control	M1/M4 Rollback to s1	WR3 Option 1
s_5		NA		
s_6	a_1-a_4	Fire	M3 Roll Forward to Abort	WR2 Option 1
s_7	a_1-a_4	Fire	M3 Roll Forward to Abort	WR2 Option 1
s_8		NA		
s_9	a_1-a_4	Fire	M3 Roll Forward to Abort	WR2 Option 1
s_{10}	a_1-a_4	Fire	M3 Roll Forward to Abort	WR2 Option 1
s_{11}	a_1-a_4	Fire	M3 Roll Forward to Abort	WR2 Option 1
s_{12}		NA		
s_{13}	a_1-a_4	Fire	M3 Roll Forward to Abort	WR2 Option 1
s_{14}	a_1-a_4	Fire	M3 Roll Forward to Abort	WR2 Option 1
s_{15}	a_1-a_4	Fire	M3 Roll Forward to Abort	WR2 Option 1
s_{16}	a_1-a_4	Fire	M3 Roll Forward to Abort	WR2 Option 1
s_{17}	a_1-a_4	Fire	M3 Roll Forward to Abort	WR2 Option 1
s_{18}		NA		
s_{19}	a_1-a_4	Fire	M3 Roll Forward to Abort	WR2 Option 1
s_{20}	a_1-a_4	Fire	M3 Roll Forward to Abort	WR2 Option 1

Table 4.3: Launch Vehicle Attack Applicability Matrix

A/CT	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	s_{13}	s_{14}	s_{15}	s_{16}	s_{17}	s_{18}	s_{19}	s_{20}	s_{21}
a_1	0	1	1	1	0	1	1	0	1	1	1	0	1	1	1	1	1	0	1	1	0
a_2	0	1	1	1	0	1	1	0	1	1	1	0	1	1	1	1	1	0	1	1	0
a_3	0	0	1	1	0	1	1	0	1	1	1	0	1	1	1	1	1	0	1	1	0
a_4	0	0	1	1	0	1	1	0	1	1	1	0	1	1	1	1	1	0	1	1	0

tion in the CEFSM. The specifics of transitions can be found in [37]. The test suite $CT = \{bt_1\}$ where $bt_1 = \{$ $s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_4, s_5, s_6, s_6, s_7, s_7, s_7, s_8, s_9, s_9, s_{10}, s_{10}, s_{11}, s_{11}, s_{11}, s_{12}, s_{13}, s_{13}, s_{14}, s_{14}, s_{15}, s_{15}, s_{16}, s_{16}, s_{17}, s_{17}, s_{17}, s_{18}, s_{19}, s_{19}, s_{20}, s_{21}\}$.

Assuming edge coverage is required, the test paths BT fulfill this requirement. By using reachability analysis, we find that these paths are feasible because there are no conflicting predicates along the paths. Some attacks are only feasible in certain states. Table 4.3 shows the attack applicability matrix AM. Some attacks are launch stage specific. We use mitigation actions to mitigate these attacks and avoid adversary behavior. We choose (p,a) pairs that meet node attack coverage criteria. Note that in this case study, A5 is not applicable as there is no GPS system on a launch vehicle.

4.2.3 Security Test Requirements

Next, we determine security test requirements (p, a) for coverage criteria 1-4 (SR_1-SR_4). As per the definition of CT in Section 4.2.1, CT is given as CT = $bt_1 = \{$ s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_4, s_5, s_6, s_6, s_7, s_7, s_7, s_8, s_9, s_9, s_{10}, s_{10}, s_{11}, s_{11}, s_{11}, s_{12}, s_{13}, s_{13}, s_{14}, s_{14}, s_{15}, s_{15}, s_{16}, s_{16}, s_{17}, s_{17}, s_{17}, s_{18}, s_{19}, s_{19}, s_{20}, $s_{21}\}$. There are 39 positions. We now apply coverage criteria for the above positions ($1 \leq s \leq 39$) and attacks ($1 \leq a \leq 4$).

Coverage Criteria 1: all states, all applicable attacks. The required (p,a) combinations are shown in Table 4.4 as 1 entries. This requires 126 security tests. While theoretically feasible, it is a large number of tests. This is not only due to the length of CT, but also due to the large proportion of 1 entries in AM, i.e. whether attacks are feasible in most states or not.

Coverage Criteria 2 and 3: all unique nodes, all applicable attacks and all tests, all unique nodes, all applicable attacks. The launch vehicle example is very sequential. Therefore, criteria 2 and criteria 3 have the same matrix, as shown in Table 4.5. They both require 52 security tests.

Coverage Criteria 4: all tests, all unique nodes, some attacks. This criteria does not require that all attacks be injected at every state even though each attack must be selected at least once. (p,a) combinations that meet this requirement are shown in Table 4.6 as 1 entries. There are 5 security tests required.

4.2.4 Mitigation Requirements

The mitigation requirements are summarized in Table 4.2. The tables list each state with possible attacks, the risks of the attack being executed as well as the mitigation required and the weaving rule we will use to mitigate the attack. If a state is not listed in the table, it is because no attack is possible in that state (i.e. idle states). For example in state 2 (s_2) or (network connection) an attacker could execute either a RF DOS attack (a_1) or a DOS Flood attack (a_2) which would result in a loss or network connection or confusing the launch conductor or vehicle. The mitigation would be to roll the system back to the idle state (s_1) and attempt to reconnect to the network.

Table 4.4: Launch Vehicle Security Test Requirements - Criteria 1

	bt_1																																						
A/CT	s_1	s_2	s_2	s_3	s_3	s_4	s_4	s_4	s_5	s_6	s_6	s_7	s_7	s_7	s_8	s_9	s_9	s_{10}	s_{10}	s_{11}	s_{11}	s_{11}	s_{12}	s_{13}	s_{13}	s_{14}	s_{14}	s_{15}	s_{15}	s_{16}	s_{16}	s_{17}	s_{17}	s_{17}	s_{18}	s_{19}	s_{19}	s_{20}	s_{21}
a_1	0	1	1	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0	0
a_2	0	1	1	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0	0
a_3	0	0	0	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0	0
a_4	0	0	0	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0	0

Table 4.5: Launch Vehicle Security Test Requirements - Criteria 2 and Criteria 3

	bt_1																																						
A/CT	s_1	s_2	s_2	s_3	s_3	s_4	s_4	s_4	s_5	s_6	s_6	s_7	s_7	s_7	s_8	s_9	s_9	s_{10}	s_{10}	s_{11}	s_{11}	s_{11}	s_{12}	s_{13}	s_{13}	s_{14}	s_{14}	s_{15}	s_{15}	s_{16}	s_{16}	s_{17}	s_{17}	s_{17}	s_{18}	s_{19}	s_{19}	s_{20}	s_{21}
a_1	0	1	0	1	0	1	0	0	0 0	1	0	1	0	0	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	1	0	1	0	0	0	1	0	0	0
a_2	0	0	0	0	0	1	0	0	0 0	1	0	1	0	0	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	1	0	1	0	0	0	1	0	0	0
a_3	0	0	0	0	0	1	0	0	0 0	1	0	1	0	0	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	1	0	1	0	0	0	1	0	0	0
a_4	0	1	0	1	0	1	0	0	0 0	1	0	1	0	0	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	1	0	1	0	0	0	1	0	0	0

Table 4.6: Launch Vehicle Security Test Requirements - Criteria 4

	bt_1																																						
A/CT	s_1	s_2	s_2	s_3	s_3	s_4	s_4	s_4	s_5	s_6	s_6	s_7	s_7	s_7	s_8	s_9	s_9	s_{10}	s_{10}	s_{11}	s_{11}	s_{11}	s_{12}	s_{13}	s_{13}	s_{14}	s_{14}	s_{15}	s_{15}	s_{16}	s_{16}	s_{17}	s_{17}	s_{17}	s_{18}	s_{19}	s_{19}	s_{20}	s_{21}
a_1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a_2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a_3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
a_4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

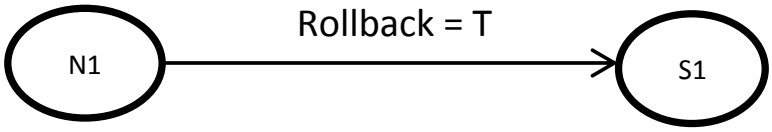

Figure 4.3: Roll Back Mitigation Model M1

Table 4.7: Security Mitigation Test for Criteria 4

Security Test	**SMT**	**bt**
smt_1	$s_1, s_2, N_1, s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_4, s_5, s_6, s_6,$ $s_7, s_7, s_7, s_8, s_9, s_9, s_{10}, s_{10}, s_{11}, s_{11}, s_{11}, s_{12}, s_{13}, s_{13},$ $s_{14}, s_{14}, s_{15}, s_{15}, s_{16}, s_{16}, s_{17}, s_{17}, s_{17}, s_{18}, s_{19}, s_{19}, s_{20}, s_{21}$	bt_1
smt_2	$s_2, s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_4, s_5, s_6, s_6, s_7,$ $s_7, s_7, s_8, s_9, N_{11}, N_{12}$	bt_1
smt_3	$s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_4, s_5, s_6, s_6, s_7,$ $s_7, s_7, s_8, s_9, s_9, s_{10}, s_{10}, s_{11}, s_{11}, s_{11}, s_{12}, s_{13}, s_{13},$ s_{14}, N_{11}, N_{12}	bt_1
smt_4	$s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_4, s_5, s_6, s_6,$ $s_7, s_7, s_7, s_8, s_9, s_9, s_{10}, s_{10}, s_{11}, s_{11}, s_{11}, s_{12}, s_{13}, s_{13},$ $s_{14}, s_{14}, s_{15}, s_{15}, s_{16}, s_{16}, s_{17}, s_{17}, s_{17}, s_{18}, s_{19}, N_{11}, N_{12}$	bt_1

4.2.5 Pre Launch System Mitigation Models

Security attacks can be injected into any state that gets a command from the launch conductor. Mitigations for this test case include a roll-back to a safe state or a roll-forward to a safe state (such as an abort command), as these are the only two feasible mitigations. Mitigation models are shown in Figure 4.3 and Figure 4.4. $MT = \{mt_1..mt_2\}$ $mt_1 = (N_1, s_1)$ $mt_2 = (N_{11}, N_{12})$

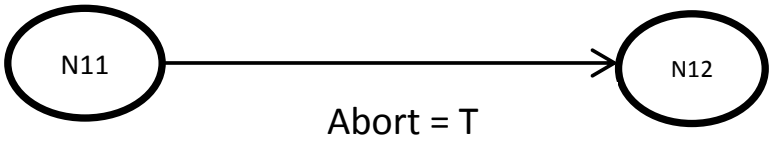

Figure 4.4: Roll Forward Mitigation Model M2

4.2.6 Pre Launch System Security Test Suite

The security test suite for a pre-launch vehicle is defined in Table 4.7. This test suite covers the tests that would be required for criteria 4 noted above. We complete the example by demonstrating executable tests are as follows:

$SMT1:$

$s_1 : (Idle, [startSequence = True], startInitialize)$

$/(NetworkConnection)$

$s_2 : (NetworkConnection.[NetworkConnection = True])$

$/send(NetworkConnectionSuccess)(HazardLightsOn,)$

$N_1 : (RollBackSafe, [AttackType = a_1][SafeState = s_1],$

$RollBackSuccess/send(RollBackSuccess)(RollBackSafe)$

$s_1 : (Idle, [startSequence = True], startInitialize)$

$/(NetworkConnection)$

$s_2 : (NetworkConnection.[NetworkConnection = True])$

$/send(NetworkConnectionSuccess)(NetworkConnection,)$

$s_2 : (NetworkConnection.[NetworkConnection = True])$

$/send(NetworkConnectionSuccess)(HazardLightsOn,)$

$s_3 : (HazardLightsOn.[HazardLightsOn = True])$

$/send(HazardLightsOnSuccess)(HazardLightsOn)$

$s_3 : (HazardLightsOn.[HazardLightsOn = True])$

$/send(HazardLightsOnSuccess)(CountdownClockReset)$

$s_4 : (CountdownClockReset.[CountdownClockReset = True])$

$/send(CountdownClockResetSuccess)(CountdownClockReset)$

$s_4 : (CountdownClockReset.[CountdownClockReset = True])$

$/send(CountdownClockResetSuccess)(Init)$

$s_5 : (Init.[startSequence = True], startECSInitialization)$

$/(AirConditioning)$

$s_6 : (AirConditioning.[AirConditioning = True])$

$/send(AirConditioningSuccess)(AirConditioning)$

$s_6 : (AirConditioning.[AirConditioning = True])$

$/send(AirConditioningSuccess)(NitrogenPurge)$

$s_7 : (NitrogenPurge.[NitrogenPurge = True])$

$/send(NitrogenPurgeSuccess)(NitrogenPurge)$

$s_7 : (NitrogenPurge.[NitrogenPurge = True])$

$/send(NitrogenPurgeSuccess)(Init)$

$s_8 : (Init.[startSequence = True], LO2Chk)/(LO2Chk)$

$s_9 : (LO2Chk.[LO2Chk = True])/send(LO2ChkSuccess)$

$(LO2Chk)$

$s_9 : (LO2Chk.[LO2Chk = True])/send(LO2ChkSuccess)$

$(HeliumChk)$

s_{10}:$(HeliumChk.[HeliumChk = True]])$

$/send(HeliumChkSuccess)(HeliumChk)$

s_{10}:$(HeliumChk.[HeliumChk = True]])$

$/send(HeliumChkSuccess)(LH2Chk)$

s_{11}:($LH2Chk[LH2Chk = True]$])

$/send(LH2ChkSuccess)(LH2Chk)$

s_{11}:($LH2Chk[LH2Chk = True]$])

$/send(LH2ChkSuccess)(Init)$

s_{12}:($Init.[startSequence = True]$,

$startPreflight)/(INSTChk)$

s_{13}:($INSTChk.[INSTChk = True]$)

$/send((INSTChkSuccess)(INSTChk)$

s_{13}:($INSTChk.[INSTChk = True]$)

$/send((INSTChkSuccess)(CryoTesting)$

s_{14}:($CryoTesting.[CryoTesting = True]$)

$/send(CryoTestingSuccess)(CryoTesting)$

s_{14}:($CryoTesting.[CryoTesting = True]$)

$/send(CryoTestingSuccess)(ChillDown)$

s_{15}:($ChillDown.[ChillDown = True]$)

$/send(ChillDownSuccess)(ChillDown)$

s_{15}:($ChillDown.[ChillDown = True]$)

$/send(ChillDownSuccess)(BatteryCheck)$

s_{16}:($BatteryCheck.[BatteryCheck = True]$)

$/send(BatteryCheckSuccess)(BatteryCheck)$

s_{16}:($BatteryCheck.[BatteryCheck = True]$)

$/send(BatteryCheckSuccess)(InitiateFueling)$

s_{17}:($InitiateFueling.[InitiateFueling = True]$)

$/send(InitiateFuelingSuccess)(InitiateFueling)$

s_{17}:($InitiateFueling.[InitiateFueling = True]$)

$/send(InitiateFuelingSuccess)(Init)$

s_{18}:$(Init.[startSequence = True], Flight)$

$/(InternalBattery)$

s_{19}:$(InternalBattery.[InternalBattery = True])$

$/send(InternalBatterySuccess)(InternalBattery)$

s_{19}:$(InternalBattery.[InternalBattery = True])$

$/send(InternalBatterySuccess)(FlightCommand)$

s_{20}:$(FlightCommand.[FlightCommand = True])$

s_{21}:$(Success.[Success = True])(EndSequence)$;

$SMT2:$

$s_1 : (Idle, [startSequence = True], startInitialize)$

$/(NetworkConnection)$

$s_2 : (NetworkConnection.[NetworkConnection = True])$

$/send(NetworkConnectionSuccess)(HazardLightsOn,)$

$N_1 : (RollBackSafe, [AttackType = a_1][SafeState = s_1],$

$RollBackSuccess/send(RollBackSuccess)(RollBackSafe)$

$s_1 : (Idle, [startSequence = True], startInitialize)$

$/(NetworkConnection)$

$s_2 : (NetworkConnection.[NetworkConnection = True])$

$/send(NetworkConnectionSuccess)(NetworkConnection,)$

$s_2 : (NetworkConnection.[NetworkConnection = True])$

$/send(NetworkConnectionSuccess)(HazardLightsOn,)$

$s_3 : (HazardLightsOn.[HazardLightsOn = True])$

$/send(HazardLightsOnSuccess)(HazardLightsOn)$

$s_3 : (HazardLightsOn.[HazardLightsOn = True])$

$/send(HazardLightsOnSuccess)(CountdownClockReset)$

$s_4 : (CountdownClockReset.[CountdownClockReset = True])$

$/send(CountdownClockResetSuccess)(CountdownClockReset)$

$s_4 : (CountdownClockReset.[CountdownClockReset = True])$

$/send(CountdownClockResetSuccess)(Init)$

$s_5 : (Init.[startSequence = True], startECSInitialization)$

$/(AirConditioning)$

$s_6 : (AirConditioning.[AirConditioning = True])$

$/send(AirConditioningSuccess)(AirConditioning)$

$s_6 : (AirConditioning.[AirConditioning = True])$

$/send(AirConditioningSuccess)(NitrogenPurge)$

$s_7 : (NitrogenPurge.[NitrogenPurge = True])$

$/send(NitrogenPurgeSuccess)(NitrogenPurge)$

$s_7 : (NitrogenPurge.[NitrogenPurge = True])$

$/send(NitrogenPurgeSuccess)(Init)$

$s_8 : (Init.[startSequence = True], LO2Chk)/(LO2Chk)$

$s_9 : (LO2Chk.[LO2Chk = True])/send(LO2ChkSuccess)$

$(LO2Chk)$

$N_1 1 : (Attack, [Attack = True][AttackType = a_3]$

$RolllForward/send(AbortMessage)(RollForwardAbort)$

$N_1 2 : (RollForwardAbort, [Abort = True],$

$AbortSuccess/send(AbortSuccess)(EndSequence);$

$SMT3 :$

$s_1 : (Idle, [startSequence = True], startInitialize)$

$/(NetworkConnection)$

$s_2 : (NetworkConnection.[NetworkConnection = True])$

$/send(NetworkConnectionSuccess)(HazardLightsOn,)$

$N_1 : (RollBackSafe, [AttackType = a_1][SafeState = s_1],$

$RollBackSuccess/send(RollBackSuccess)(RollBackSafe)$

$s_1 : (Idle, [startSequence = True], startInitialize)$

$/(NetworkConnection)$

$s_2 : (NetworkConnection.[NetworkConnection = True])$

$/send(NetworkConnectionSuccess)(NetworkConnection,)$

$s_2 : (NetworkConnection.[NetworkConnection = True])$

$/send(NetworkConnectionSuccess)(HazardLightsOn,)$

$s_3 : (HazardLightsOn.[HazardLightsOn = True])$

$/send(HazardLightsOnSuccess)(HazardLightsOn)$

$s_3 : (HazardLightsOn.[HazardLightsOn = True])$

$/send(HazardLightsOnSuccess)(CountdownClockReset)$

$s_4 : (CountdownClockReset.[CountdownClockReset = True])$

$/send(CountdownClockResetSuccess)(CountdownClockReset)$

$s_4 : (CountdownClockReset.[CountdownClockReset = True])$

$/send(CountdownClockResetSuccess)(Init)$

$s_5 : (Init.[startSequence = True], startECSInitialization)$

$/(AirConditioning)$

$s_6 : (AirConditioning.[AirConditioning = True])$

$/send(AirConditioningSuccess)(AirConditioning)$

$s_6 : (AirConditioning.[AirConditioning = True])$

$/send(AirConditioningSuccess)(NitrogenPurge)$

$s_7 : (NitrogenPurge.[NitrogenPurge = True])$

$/send(NitrogenPurgeSuccess)(NitrogenPurge)$

$s_7 : (NitrogenPurge.[NitrogenPurge = True])$

$/send(NitrogenPurgeSuccess)(Init)$

$s_8 : (Init.[startSequence = True], LO2Chk)/(LO2Chk)$

$s_9 : (LO2Chk.[LO2Chk = True])/send(LO2ChkSuccess)$

$(LO2Chk)$

$s_9 : (LO2Chk.[LO2Chk = True])/send(LO2ChkSuccess)$

$(HeliumChk)$

s_{10}:$(HeliumChk.[HeliumChk = True]])$

$/send(HeliumChkSuccess)(HeliumChk)$

s_{10}:$(HeliumChk.[HeliumChk = True]])$

$/send(HeliumChkSuccess)(LH2Chk)$

s_{11}:$(LH2Chk[LH2Chk = True]])$

$/send(LH2ChkSuccess)(LH2Chk)$

s_{11}:$(LH2Chk[LH2Chk = True]])$

$/send(LH2ChkSuccess)(Init)$

s_{12}:$(Init.[startSequence = True],$

$startPreflight)/(INSTChk)$

s_{13}:$(INSTChk.[INSTChk = True])$

$/send((INSTChkSuccess)(INSTChk)$

s_{13}:$(INSTChk.[INSTChk = True])$

$/send((INSTChkSuccess)(CryoTesting)$

s_{14}:$(CryoTesting.[CryoTesting = True])$

$/send(CryoTestingSuccess)(CryoTesting)$

$N_1 1 : (Attack, [Attack = True][AttackType = a_3]$

$RolllForward/send(AbortMessage)(RollForwardAbort)$

$N_1 2 : (RollForwardAbort, [Abort = True],$

$AbortSuccess/send(AbortSuccess)(EndSequence);$

$SMT4 :$

$s_1 : (Idle, [startSequence = True], startInitialize)$

$/(NetworkConnection)$

$s_2 : (NetworkConnection.[NetworkConnection = True])$

$/send(NetworkConnectionSuccess)(HazardLightsOn,)$

$N_1 : (RollBackSafe, [AttackType = a_1][SafeState = s_1],$

$RollBackSuccess/send(RollBackSuccess)(RollBackSafe)$

$s_1 : (Idle, [startSequence = True], startInitialize)$

$/(NetworkConnection)$

$s_2 : (NetworkConnection.[NetworkConnection = True])$

$/send(NetworkConnectionSuccess)(NetworkConnection,)$

$s_2 : (NetworkConnection.[NetworkConnection = True])$

$/send(NetworkConnectionSuccess)(HazardLightsOn,)$

$s_3 : (HazardLightsOn.[HazardLightsOn = True])$

$/send(HazardLightsOnSuccess)(HazardLightsOn)$

$s_3 : (HazardLightsOn.[HazardLightsOn = True])$

$/send(HazardLightsOnSuccess)(CountdownClockReset)$

$s_4 : (CountdownClockReset.[CountdownClockReset = True])$

$/send(CountdownClockResetSuccess)(CountdownClockReset)$

$s_4 : (CountdownClockReset.[CountdownClockReset = True])$

$/send(CountdownClockResetSuccess)(Init)$

$s_5 : (Init.[startSequence = True], startECSInitialization)$

$/(AirConditioning)$

$s_6 : (AirConditioning.[AirConditioning = True])$

$/send(AirConditioningSuccess)(AirConditioning)$

$s_6 : (AirConditioning.[AirConditioning = True])$

$/send(AirConditioningSuccess)(NitrogenPurge)$

$s_7 : (NitrogenPurge.[NitrogenPurge = True])$

$/send(NitrogenPurgeSuccess)(NitrogenPurge)$

$s_7 : (NitrogenPurge.[NitrogenPurge = True])$

$/send(NitrogenPurgeSuccess)(Init)$

$s_8 : (Init.[startSequence = True], LO2Chk)/(LO2Chk)$

$s_9 : (LO2Chk.[LO2Chk = True])/send(LO2ChkSuccess)$

$(LO2Chk)$

$s_9 : (LO2Chk.[LO2Chk = True])/send(LO2ChkSuccess)$

$(HeliumChk)$

s_{10}:$(HeliumChk.[HeliumChk = True]])$

$/send(HeliumChkSuccess)(HeliumChk)$

s_{10}:$(HeliumChk.[HeliumChk = True]])$

$/send(HeliumChkSuccess)(LH2Chk)$

s_{11}:$(LH2Chk[LH2Chk = True]])$

$/send(LH2ChkSuccess)(LH2Chk)$

s_{11}:$(LH2Chk[LH2Chk = True]])$

$/send(LH2ChkSuccess)(Init)$

s_{12}:$(Init.[startSequence = True],$

$startPreflight)/(INSTChk)$

s_{13}:$(INSTChk.[INSTChk = True])$

$/send((INSTChkSuccess)(INSTChk)$

s_{13}:$(INSTChk.[INSTChk = True])$

$/send((INSTChkSuccess)(CryoTesting)$

s_{14}:$(CryoTesting.[CryoTesting = True])$

$/send(CryoTestingSuccess)(CryoTesting)$

s_{14}:$(CryoTesting.[CryoTesting = True])$

$/send(CryoTestingSuccess)(ChillDown)$

s_{15}:$(ChillDown.[ChillDown = True])$

$/send(ChillDownSuccess)(ChillDown)$

s_{15}:$(ChillDown.[ChillDown = True])$

$/send(ChillDownSuccess)(BatteryCheck)$

s_{16}:$(BatteryCheck.[BatteryCheck = True])$

$/send(BatteryCheckSuccess)(BatteryCheck)$

s_{16}:$(BatteryCheck.[BatteryCheck = True])$

$/send(BatteryCheckSuccess)(InitiateFueling)$

s_{17}:$(InitiateFueling.[InitiateFueling = True])$

$/send(InitiateFuelingSuccess)(InitiateFueling)$

s_{17}:$(InitiateFueling.[InitiateFueling = True])$

$/send(InitiateFuelingSuccess)(Init)$

s_{18}:$(Init.[startSequence = True], Flight)$

$/(InternalBattery)$

s_{19}:$(InternalBattery.[InternalBattery = True])$

$/send(InternalBatterySuccess)(InternalBattery)$

$N_1 1 : (Attack, [Attack = True][AttackType = a_3]$

$RolllForward/send(AbortMessage)(RollForwardAbort)$

$N_1 2 : (RollForwardAbort, [Abort = True],$

$AbortSuccess/send(AbortSuccess)(EndSequence)$;

4.2.7 Discussion

We have demonstrated that our approach can be applied to a pre-flight launch vehicle. We have also demonstrated that we can categorize attacks into more and less severe, depending on when they occur in the launch sequence. In the early stages of the launch, we can roll-back and retry various commands, but when the cryogenic fuels start being deployed, it becomes much more difficult to mitigate an attack and in most cases a controlled abort is the only option. We have demonstrated a phased approach using a single test path. Our approach has limited actions including: abort and safe state. Our approach has shown that we can test the security of a pre-flight launch vehicle with a very small security test suite. We demonstrate that we can leverage an MBT behavioral test suite to build a security test suite, using CEFSM models. We show that we can model required mitigations for security attacks and generate mitigation tests. We can identify criteria for covering attack scenarios in a systematic way. We can also use behavioral tests, mitigation test and attack scenarios to build a systematic approach and build our security testing (MBST). It is a scalable approach that can be applied to a pre-launch vehicle.

4.3 Case Study 2: Post Launch System

Our approach can also be demonstrated on a launch vehicle post-liftoff. The goal of this case study is to expand our previous case study to include states after the vehicle has left the launch pad. After the launch vehicle has jettisoned off the launch pad, an attack could have serious consequences that could not only include loss of the vehicle or loss of life, but could be redirected for malicious intent or intentionally misused to target and explode a high profile location such as the white house or a military facility. We will demonstrate that we can leverage an MBT behavioral test

suite to build a security test suite, using CEFSM models and that our approach can be applied to a post-launch vehicle. The launch vehicle for this example is a two stage launch vehicle which consists of a stack including the:

- booster - Generally a solid rocket motor which jettisons the vehicle into orbit.
- upper stage - A smaller engine capable of maneuvering the payload into the proper position.
- payload - A Satellite or Item that is intended to be delivered into orbit generally a science instrument or a military communication device.

The behavioral model of the post launch vehicle in Communicating Extended Finite State Machine (CEFSM) format is shown in Figures 4.5 and 4.6. At the time of launch, the vehicle transitions (t_1) to the booster fire (s_1) and the vehicle lifts off of the launch pad. The vehicle transitions (t_2) and initiates a Safe Arm Device (SAD)(s_3). One and half minutes into the flight, the booster (typically a Solid Rocket Motor SRM) is commanded to separate (s_4) from the vehicle. Four minutes into flight, a payload fairing jettison (s_6) command is sent to release the nose cone (payload protective cover), which exposes the payload (s_7). Seven minutes into the flight, a fire command (s_9) is sent and the first stage separates (s_{10}) from the vehicle. When the vehicle reaches LEO, another fire (s_{12}) command is sent and the first stage is separated (s_{13})from the payload. Finally the payload is injected into orbit(s_{14}). If all goes as expected, the mission is complete. If an error is detected before the booster is separated, the launch conductor may initiate an abort using a safe arm device (SAD). The SAD forces the SRM to ignite, which terminates the flight by exploding the solid rocket motor. Once the vehicle has left the launch pad, the only option to abort the mission is to destroy the vehicle and the payload.

4.3.1 Post Launch System Behavioral Model

There are 14 unique states and 25 transitions in the model. States and transitions define the test paths. The specifics of transitions can be found in [37].

The test suite $CT = \{bt_1\}$ where

$bt_1 = \{s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_4, s_5, s_6, s_6, s_7, s_7, s_7, s_8, s_9, s_9, s_{10}, s_{10}, s_{10}, s_{11}, s_{12}, s_{12}, s_{13}, s_{13}, s_{14}, s_{14}, s_{14}\}$. The paths are feasible based on reachability analysis, as there are no conflicting predicates between transitions.

4.3.2 Security Attacks on a Post Launch System

Security attacks with the post launch vehicle can be injected at each state where there is a communication with the ground system. DOS RFJAM and DOS Flood commands are possible in every state except the idle states. Spoofed and MITM commands are also applicable in all states except idle as each has a communication path to the launch conductor or ground system. DOS attacks to the vehicle can be rolled back and reattempted. However, spoofed commands that can separate key elements of the vehicle must result in an abort. The attacks that are possible in a post-flight system are defined in Table 4.8. Note that A5, the GPS JAM attack is not included, because GPS is not used in a launch vehicle.

Table 4.8: Possible Post-flight Launch Vehicle Attacks

Attack	**Type**	**Description**
A_1	DOS RFJam	Floods RF Channels
A_2	DOS Flood	Floods Comm Channels
A_3	Sniff & Spoof	Listens then injects Packets
A_4	Sniff & MITM	Listens, Forwards and Injects Packets

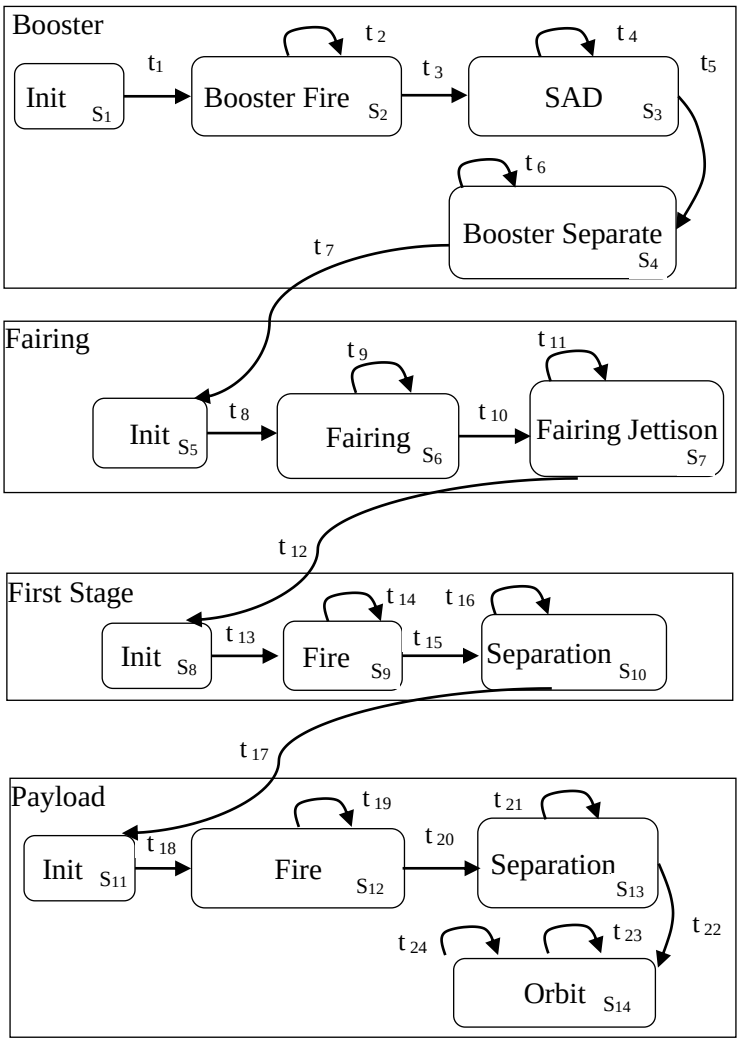

Figure 4.5: Post Launch Vehicle CEFSM Part 1

t1:(Init, [startSequence=True],startLaunch)/(BoosterFire)

t2:(BoosterFire. [BoosterFire =False])/send(BoosterFireFailure)
(BoosterFire)

t3:(BoosterFire. [BoosterFire =True])/send(BoosterFireSuccess) (SAD)

t4:(SAD. [SAD =False])/send(SADFailure) (SAD)

t5:(SAD. [SAD =True])/send(SADSuccess) (BoosterSep)

t6:(BoosterSep. [BoosterSep =False])/send(BoosterFireFailure)
(BoosterSep)

t7:(BoosterSep. [BoosterSep =True])/send(BoosterFireSuccess)

t8:(Init. [startSequence=True],startFairing)/ (Fairing)

t9:(Fairing. [Fairing =False)/send(FairingFailure) (Fairing)

t10:(Fairing. [Fairing =True])/send(FairingSuccess)(Jettison)

t11:(Jettison. [JettisonSep =False])/send(JettisonSepFailure) (Jettison)

t12:(Jettison. [JettisonSep =True])/send(JettisonSepSuccess)

t13: (Init. [startSequence=True],startFS)/ (FSFire)

t14:(FSFire. [FSFire =False])/send(FSFireFailure)(FSFire)

t15:(FSFire. [FSFire =True])/send(FSFireFailure) (FSSeparation)

t16:(FSSeparation. [FSSeparation =False])/send(FSSeparationFailure)
(FSSeparation)

t17:(FSSeparation. [FSSeparation =True]])/send(FSSeparationSuccess)

t18: (Init. [startSequence=True],startPL)/ (PLFire)

t19:(PLFire. [PLFire =False])/send(PLFireFailure) (PLFire)

t20:(PLFire. [PLFire =True])/send((PLFireSuccess)(PLSeparation)

t21:(PLSeparation.
[PLSeparation=False])/send(PLSeparationFailure)(PLSeparation)

t22:(PLSeparation. [PLSeparation
=True])/send(PLSeparationSuccess)(Orbit)

t23:(Orbit. [Orbit =False])/send(OrbitFailure)(Orbit)

t24:(Orbit. [Orbit =True])/send(OrbitSuccess)

Figure 4.6: Post Launch Vehicle CEFSM Part 2

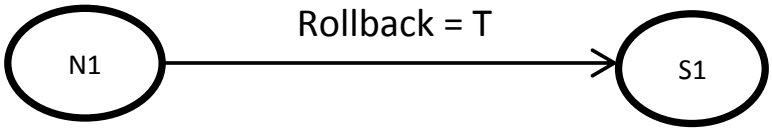

Figure 4.7: Roll Forward Mitigation Model

4.3.3 Post Launch System Mitigation Models

Security attacks can be injected into any state that gets a command from another stage or from the ground system. Any attack on a post-liftoff launch vehicle has the potential to be catastrophic. Mitigations in this case include an abort command or rollback to a safe state, as these are the only two feasible mitigations post-launch. Mitigation models are shown in Figure 4.7 and Figure 4.8.

$MT = \{mt_1..mt_2\}$

$mt_1 = (N_1, s_1)$

$mt_3 = (N_{11}, N_{12})$

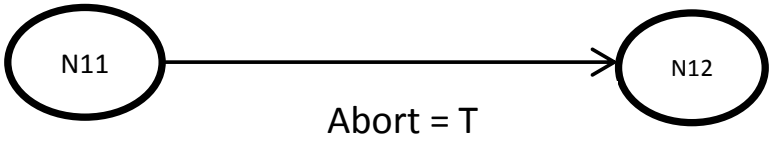

Figure 4.8: Roll Back Mitigation Model

Table 4.9: Post Launch Test Attacks

State	Attack	Risk if Attack Successful	MT Mitigation	WR
s1		NA		
s2	A1-A4	Command Spoofed	M1 Roll Back s1	WR3 Option 1
s3	A3-A4	Command Spoofed	M1 Roll Back s1	WR3 Option 1
s4	A3-A4	Malicious Command	M4 Rollforward Abort	WR2 Option 1
s5		NA		
s6	A1-A4	Command Spoofed	M4 Rollforward Abort	WR2 Option 1
s7	A3-A4	Malicious Command	M4 Rollforward Abort	WR2 Option 1
s8		NA		
s9	A1-A4	Command Spoofed	M4 Rollforward Abort	WR2 Option 1
s10	A3-A4	Command Spoofed	M4 Rollforward Abort	WR2 Option 1
s11		NA		
s12	A1-A4	Command Spoofed	M4 Rollforward Abort	WR2 Option 1
s13	A3-A4	Malicious Command	M4 Rollforward Abort	WR2 Option 1
s14	A3-A4	Malicious Command	M4 Rollforward Abort	WR2 Option 1

4.3.4 Security Test Requirements (Attack Scenarios)

Mitigation requirements are summarized in Table 4.9. The tables list each state with possible attacks, the risks of the attack being executed as well as the mitigation required and the weaving rule we will use to mitigate the attack. If a state is not listed in the table, it is because no attack is possible in that state (i.e. idle states). For example in state 2 (s_2) or (booster command sniffed) an attacker could attempt to execute either a RF DOS attack (A1) or a DOS Flood attack (A2) which would result in the booster not getting its command. The mitigation would be to roll the system back to the idle state (s_1) and retry the booster command.

Table 4.10: Post Launch Vehicle Attack Applicability Matrix

A/CT	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	s_{13}	s_{14}
a_1	0	1	1	1	0	1	1	0	1	1	0	1	1	1
a_2	0	1	1	1	0	1	1	0	1	1	0	1	1	1
a_3	0	1	1	1	0	1	1	0	1	1	0	1	1	1
a_4	0	1	1	1	0	1	1	0	1	1	0	1	1	1

4.3.5 Post Launch System Security Attacks and Attack Applicability

We define I $=\sum_{i=1}^{4} len(t_i)$ states s to select for attack a. $CT = \{s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_4, s_5, s_6, s_6, s_7, s_7, s_7, s_8, s_9, s_9, s_{10}, s_{10}, s_{10}, s_{11}, s_{12}, s_{12}, s_{13}, s_{13}, s_{14}, s_{14}, s_{14}\}$

There are 27 positions in this model. We now apply coverage criteria for the above positions ($1 \leq p \leq 39$) and attacks ($1 \leq a \leq 4$).

Coverage Criteria 1: all states, all applicable attacks. The required (p, a) combinations are shown in Table 4.11 as 1 entries. This requires 92 tests. This approach is practical but would not be scaleable in a larger model. Criteria 1 depends on the size of the test suite, the number of attack types, and where the attacks can occur.

Coverage Criteria 2 and 3: all unique nodes, all applicable attacks and all tests, all unique nodes, all applicable attacks. Demonstrated in Table 4.12 requires 40 tests.

Coverage Criteria 4: all tests, all unique nodes, some attacks. This criteria only injects attacks at one unique test point. Demonstrated in Table 4.13 requires 4 tests.

Table 4.11: Post Launch Security Attack Scenarios- Criteria 1

	bt_1																									
A/CT	s_1	s_2	s_2	s_3	s_3	s_4	s_4	s_5	s_6	s_6	s_7	s_7	s_7	s_8	s_9	s_9	s_{10}	s_{10}	s_{10}	s_{11}	s_{12}	s_{12}	s_{13}	s_{13}	s_{14}	s_{14}	s_{14}
a_1	0	1	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	1	1	1
a_2	0	1	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	1	1	1
a_3	0	1	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	1	1	1
a_4	0	1	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	1	1	1

Table 4.12: Post Launch Security Attack Scenarios- Criteria 2 and Criteria 3

	bt_1																									
A/CT	s_1	s_2	s_2	s_3	s_3	s_4	s_4	s_5	s_6	s_6	s_7	s_7	s_7	s_8	s_9	s_9	s_{10}	s_{10}	s_{10}	s_{11}	s_{12}	s_{12}	s_{13}	s_{13}	s_{14}	s_{14}	s_{14}
a_1	0	1	0	1	0	1	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	1	0	0
a_2	0	1	0	1	0	1	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	1	0	0
a_3	0	1	0	1	0	1	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	1	0	0
a_4	0	1	0	1	0	1	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	1	0	0

Table 4.13: Post Launch Security Attack Scenarios- Criteria 4

	bt_1																									
A/CT	s_1	s_2	s_2	s_3	s_3	s_4	s_4	s_5	s_6	s_6	s_7	s_7	s_7	s_8	s_9	s_9	s_{10}	s_{10}	s_{10}	s_{11}	s_{12}	s_{12}	s_{13}	s_{13}	s_{14}	s_{14}	s_{14}
a_1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a_2	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a_3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
a_4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0

Table 4.14: Security Mitigation Test for Criteria 4

Security Test	**SMT**	**bt**
smt_1	$s_1, s_2, N_1, s_1, s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_4, s_5, s_6, s_6, s_7, s_7, s_7,$ $s_8, s_9, s_9,\ s_{10},\ s_{10},\ s_{10},\ s_{11},\ s_{12},\ s_{12},\ s_{13},\ s_{13},\ s_{14},\ s_{14},\ s_{14}$	bt_1
smt_2	$s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_4, s_5, s_6, s_6, N_{11}, N_{12}$	bt_1
smt_3	$s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_4, s_5, s_6, s_6, s_7, s_7, s_7, s_8, s_9, s_9,$ $s_{10},\ N_{11},\ N_{12}$	bt_1
smt_4	$s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_4, s_5, s_6, s_6, s_7, s_7, s_7, s_8, s_9, s_9,$ $s_{10},\ s_{10},\ s_{10},\ s_{11},\ s_{12},\ s_{12},\ s_{13},\ s_{13},\ N_{11},\ N_{12}$	bt_1

4.3.6 Post Launch System Security Test Suite

The security test suite for a post launch vehicle is defined in Table 4.14. This test suite covers the tests that would be required for criteria 4 noted above. Eleven tests are required to provide adequate coverage. We complete the example by demonstrating executable tests are as follows:

$SMT1:$

$s_1 : (Init, [startSequence = True], startLaunch)$

$/(BoosterFire)$

$s_2 : (BoosterFire.[BoosterFire = True])$

$/send(BoosterFireSuccess)(BoosterFire)$

$N_1 : (RollBackSafe, [AttackType = a_1][SafeState = s_1],$

$RollBackSuccess/send(RollBackSuccess)(RollBackSafe)$

$s_1 : (Init, [startSequence = True], startLaunch)$

$/(BoosterFire)$

$s_2 : (BoosterFire.[BoosterFire = True])$

$/send(BoosterFireSuccess)(BoosterFire)$

$s_2 : (BoosterFire.[BoosterFire = True])$

$/send(BoosterFireSuccess)(SAD)$

$s_3 : (SAD.[SAD = True])/send(SADSuccess)$

(SAD)

$s_3 : (SAD.[SAD = True])/send(SADSuccess)$

$(BoosterSep)$

$s_4 : (BoosterSep.[BoosterSep = True])$

$/send(BoosterFireSuccess)(BoosterSep)$

$s_4 : (BoosterSep.[BoosterSep = True])$

$/send(BoosterFireSuccess)(Init)$

$s_5 : (Init.[startSequence = True], startFairing)$

$/(Fairing)$

$s_6 : (Fairing.[Fairing = True])$

$/send(FairingSuccess)(Fairing)$

$s_6 : (Fairing.[Fairing = True])$

$/send(FairingSuccess)(Jettison)$

$s_7(Jettison.[JettisonSep = True])$

$/send(JettisonSepSuccess)(Jettison)$

$s_7(Jettison.[JettisonSep = True])$

$/send(JettisonSepSuccess)(Init)$

$s_8 : (Init.[startSequence = True], startFS)$

$/(FSFire)$

$s_9 : (FSFire.[FSFire = True])$

$/send(FSFireFailure)(FSFire)$

$s_9 : (FSFire.[FSFire = True])$

$/send(FSFireFailure)(FSSeparation)$

s_{10}:$(FSSeparation.[FSSeparation = True]])$

$/send(FSSeparationSuccess)(FSSeparation)$

s_{10}:$(FSSeparation.[FSSeparation = True]])$

$/send(FSSeparationSuccess)(Init)$

s_{11}:$(Init.[startSequence = True], startPL)$

$/(PLFire)$

s_{12}:$(PLFire.[PLFire = True])$

$/send((PLFireSuccess)(PLFire)$

s_{12}:$(PLFire.[PLFire = True])$

$/send((PLFireSuccess)(PLSeparation)$

s_{13}:$(PLSeparation.[PLSeparation = True])$

$/send(PLSeparationSuccess)(PLSeparation)$

s_{13}:$(PLSeparation.[PLSeparation = True])$

$/send(PLSeparationSuccess)(Orbit)$

s_{14}:$(Orbit.[Orbit = True])$

$/send(OrbitSuccess)(EndSequence)$;

$SMT2:$

$s_1 : (Init, [startSequence = True], startLaunch)$

$/(BoosterFire)$

$s_2 : (BoosterFire.[BoosterFire = True])$

$/send(BoosterFireSuccess)(BoosterFire)$

$s_2 : (BoosterFire.[BoosterFire = True])$

$/send(BoosterFireSuccess)(SAD)$

$s_3 : (SAD.[SAD = True])/send(SADSuccess)$

(SAD)

$s_3 : (SAD.[SAD = True])/send(SADSuccess)$

$(BoosterSep)$

$s_4 : (BoosterSep.[BoosterSep = True])$

$/send(BoosterFireSuccess)(BoosterSep)$

$s_4 : (BoosterSep.[BoosterSep = True])$

$/send(BoosterFireSuccess)(Init)$

$s_5 : (Init.[startSequence = True], startFairing)$

$/(Fairing)$

$s_6 : (Fairing.[Fairing = True])$

$/send(FairingSuccess)(Fairing)$

$s_6 : (Fairing.[Fairing = True])$

$N_1 1 : (Attack, [Attack = True][AttackType = a_3]$

$RolllForward/send(AbortMessage)(RollForwardAbort)$

$N_1 2 : (RollForwardAbort, [Abort = True],$

$AbortSuccess/send(AbortSuccess)(EndSequence);$

$SMT3 :$

$s_1 : (Init, [startSequence = True], startLaunch)$

$/(BoosterFire)$

$s_2 : (BoosterFire.[BoosterFire = True])$

$/send(BoosterFireSuccess)(BoosterFire)$

$s_2 : (BoosterFire.[BoosterFire = True])$

$/send(BoosterFireSuccess)(SAD)$

$s_3 : (SAD.[SAD = True])/send(SADSuccess)$

(SAD)

$s_3 : (SAD.[SAD = True])/send(SADSuccess)$

$(BoosterSep)$

$s_4 : (BoosterSep.[BoosterSep = True])$

$/send(BoosterFireSuccess)(BoosterSep)$

$s_4 : (BoosterSep.[BoosterSep = True])$

$/send(BoosterFireSuccess)(Init)$

$s_5 : (Init.[startSequence = True], startFairing)$

$/(Fairing)$

$s_6 : (Fairing.[Fairing = True])$

$/send(FairingSuccess)(Fairing)$

$s_6 : (Fairing.[Fairing = True])$

$/send(FairingSuccess)(Jettison)$

$s_7(Jettison.[JettisonSep = True])$

$/send(JettisonSepSuccess)(Jettison)$

$s_7(Jettison.[JettisonSep = True])$

$/send(JettisonSepSuccess)(Init)$

$s_8 : (Init.[startSequence = True], startFS)$

$/(FSFire)$

$s_9 : (FSFire.[FSFire = True])$

$/send(FSFireFailure)(FSFire)$

$s_9 : (FSFire.[FSFire = True])$

$/send(FSFireFailure)(FSSeparation)$

s_{10}:$(FSSeparation.[FSSeparation = True]])$

$/send(FSSeparationSuccess)(FSSeparation)$

$N_1 1 : (Attack, [Attack = True][AttackType = a_3]$

$RolllForward/send(AbortMessage)(RollForwardAbort)$

$N_1 2 : (RollForwardAbort, [Abort = True],$

$AbortSuccess/send(AbortSuccess)(EndSequence);$

$SMT4 :$

$s_1 : (Init, [startSequence = True], startLaunch)$

$/(BoosterFire)$

$s_2 : (BoosterFire.[BoosterFire = True])$

$/send(BoosterFireSuccess)(BoosterFire)$

$s_2 : (BoosterFire.[BoosterFire = True])$

$/send(BoosterFireSuccess)(SAD)$

$s_3 : (SAD.[SAD = True])/send(SADSuccess)$

(SAD)

$s_3 : (SAD.[SAD = True])/send(SADSuccess)$

$(BoosterSep)$

$s_4 : (BoosterSep.[BoosterSep = True])$

$/send(BoosterFireSuccess)(BoosterSep)$

$s_4 : (BoosterSep.[BoosterSep = True])$

$/send(BoosterFireSuccess)(Init)$

$s_5 : (Init.[startSequence = True], startFairing)$

$/(Fairing)$

$s_6 : (Fairing.[Fairing = True])$

$/send(FairingSuccess)(Fairing)$

$s_6 : (Fairing.[Fairing = True])$

$/send(FairingSuccess)(Jettison)$

$s_7(Jettison.[JettisonSep = True])$

$/send(JettisonSepSuccess)(Jettison)$

$s_7(Jettison.[JettisonSep = True])$

$/send(JettisonSepSuccess)(Init)$

$s_8 : (Init.[startSequence = True], startFS)$

$/(FSFire)$

$s_9 : (FSFire.[FSFire = True])$

$/send(FSFireFailure)(FSFire)$

$s_9 : (FSFire.[FSFire = True])$

$/send(FSFireFailure)(FSSeparation)$

s_{10}:$(FSSeparation.[FSSeparation = True]])$

$/send(FSSeparationSuccess)(FSSeparation)$

s_{10}:$(FSSeparation.[FSSeparation = True]])$

$/send(FSSeparationSuccess)(Init)$

s_{11}:$(Init.[startSequence = True], startPL)$

$/(PLFire)$

s_{12}:$(PLFire.[PLFire = True])$

$/send((PLFireSuccess)(PLFire)$

s_{12}:$(PLFire.[PLFire = True])$

$/send((PLFireSuccess)(PLFire)$

s_{13}:$(PLSeparation.[PLSeparation = True])$

$/send(PLSeparationSuccess)(PLSeparation)$

s_{13}:$(PLSeparation.[PLSeparation = True])$

$/send(PLSeparationSuccess)(Orbit)$

$N_1 1 : (Attack, [Attack = True][AttackType = a_3]$

$RolllForward/send(AbortMessage)(RollForwardAbort)$

$N_1 2 : (RollForwardAbort, [Abort = True],$

$AbortSuccess/send(AbortSuccess)(EndSequence);$

4.3.7 Discussion

We have demonstrated that while a launch vehicle's flight only lasts for a matter of 10 to 15 minutes, the ability of an attack to compromise the system could be disastrous. We have also shown that we can mitigate most DOS attacks on a post launch vehicle by rolling back and retrying. We have also shown that on the most critical spoofing attacks, we can inject a roll forward to abort the launch. While this is extreme, it prevents the vehicle from being used for malicious purposes. This case study also demonstrated our approach's ability to be applied to multiple vehicles. Our approach has shown that we can test the security of a post-flight launch vehicle with a very small security test suite. We demonstrate that we can leverage an MBT behavioral test suite to build a security test suite, using CEFSM models. We show that we can model required mitigations for security attacks and generate mitigation tests. We can identify criteria for covering attack scenarios in a systematic way. We can also use behavioral tests, mitigation test and attack scenarios to build a systematic approach and build our security test (MBST). It is a scalable approach that can be applied to a post-launch vehicle.

4.4 UAV Case Study

4.4.1 Description of a UAV System

In this section we demonstrate our method using a UAV model. The goal of this case study is to expand our previous case studies by applying our approach to an entirely different vehicle as well as adding a new attack (GPS). The UAV launch system contains a semi-autonomous air vehicle and a ground operator. The vehicle stays in contact with the ground operator to receive flight commands. We will demonstrate that we can leverage an MBT behavioral test suite to build a security test suite, using

CEFSM models and that our approach can easily be applied to a UAV and mitigate security attacks.

The behavioral model of the UAV in Communicating Extended Finite State Machine (CEFSM) format is shown in Figures 4.9 and 4.10. The first phase of the UAV flight is to perform a series of pre-arm checks which verifies the physical state of the aircraft. It must be inspected for readiness. These checks include a battery check(s_2), a propulsion check(s_3), then finally a sensor check(s_4). When the checks are complete the system transitions with (t_7) into the second phase which is called arming. The arming phase provides power to the propulsion system. Arming may only occur in a manual or stabilized mode. The system transitions between a mode check(s_6), a home position check (s_7) (which registers the GPS coordinates of the current location of the vehicle), verifies the propulsion has been armed (s_8) as well as performing a final arm check(s_9). The third phase is a series of preflight checks that include a manual control surface check (s_{11}) (to register if the UAV can be controlled manually if necessary), a stabilized control surface check(s_{12}) (to verify the stability of the vehicle) as well as a throttle check(s_{13}). The system then transitions to a launch phase in which it performs an environment check (s_{15}) as well as an abort area clear(s_{16}). The system then transitions to a flight mode in which it performs a radio range check (s_{18}) to verify it can communicate with the ground operator and accepts flight commands (s_{19}) to take off and navigate or land. Preventing security attacks in a UAV are very important as attacks could cause the UAV to be used for malicious purposes or end up in the enemy's hands.

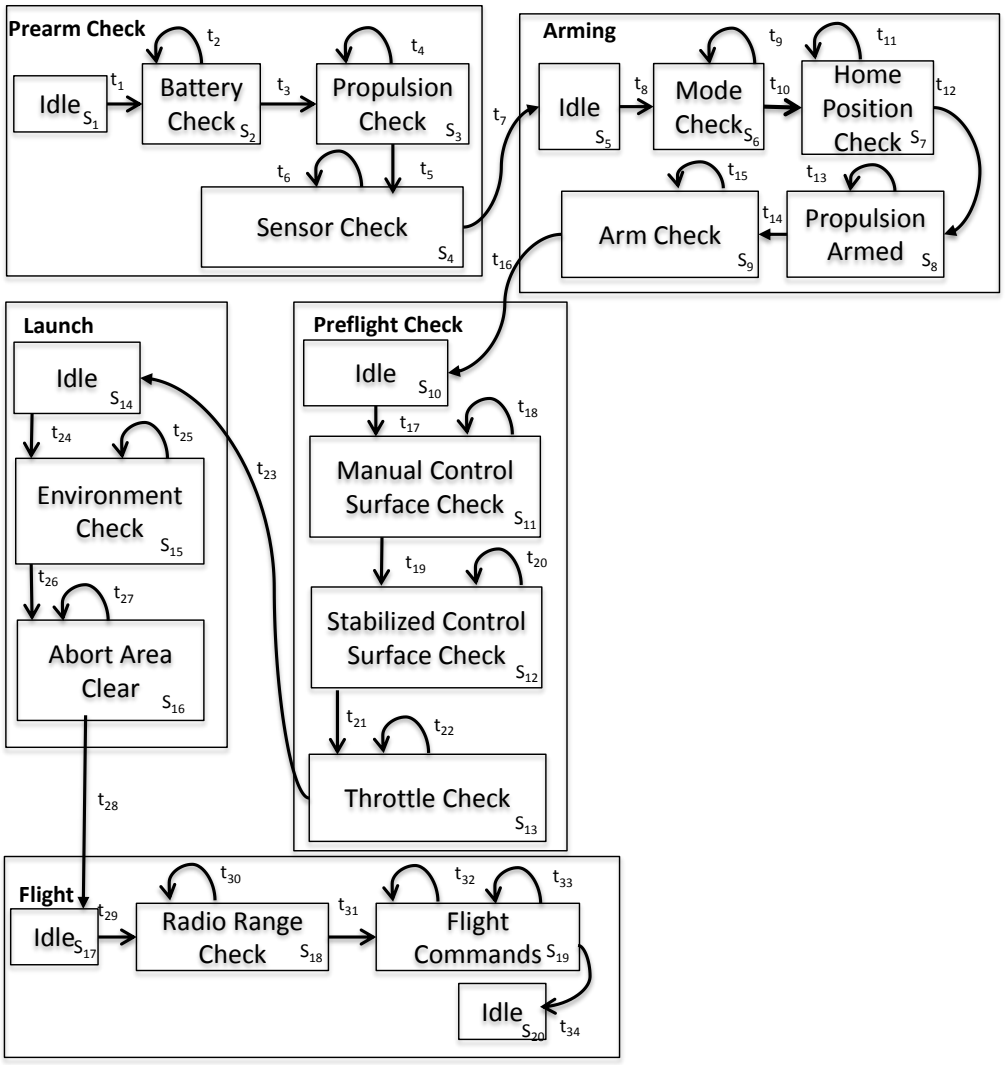

Figure 4.9: UAV CEFSM

t1:(Init, [startSequence=True],startPrearm)/(BatteryCheck)
t2:(BatteryCheck. [BatteryCheck =False])/send(BatteryCheckFailure)(BatteryCheck)
t3:(BatteryCheck. [BatteryCheck =True])/send(BatteryCheck Success) (PropCheck)
t4:(PropCheck. [PropCheck =False])/send(PropCheck Failure) (PropCheck)
t5:(PropCheck. [PropCheck =True])/send(PropCheck Success) (SensorCheck)
t6:(SensorCheck. [SensorCheck =False])/send(SensorCheckFailure) (SensorCheck)
t7:(SensorCheck. [SensorCheck =True])/send(SensorCheck Success)
t8:(Init. [startSequence=True],startArming)/ (ModeCheck)
t9:(ModeCheck. [ModeCheck =False)/send(ModeCheck Failure) (ModeCheck)
t10:(ModeCheck. [ModeCheck =True])/send(ModeCheck Success)(HomePosition)
t11:(HomePosition. [HomePosition =False])/send(HomePositionFailure) (HomePosition)
t12:(HomePosition. [HomePosition =True])/send(HomePositionSuccess) (PropArmed)
t13:(PropArmed. [PropArmed =False])/send(PropArmedFailure) (PropArmed)
t14:(PropArmed. [PropArmed =True])/send(PropArmed Success) (ArmCheck)
t15:(ArmCheck. [ArmCheck =False])/send(ArmCheckFailure) (ArmCheck)
t16:(ArmCheck. [ArmCheck =True])/send(ArmCheckSuccess)
t17:(Init. [startSequence=True],Preflight)/ (ManCtlSurfCheck)
t18:(ManCtlSurfCheck. [ManCtlSurfCheck =False])/send(ManCtlSurfCheck)(ManCtlSurfCheck)
t19:(ManCtlSurfCheck. [ManCtlSurfCheck =True])/send(ManCtlSurfCheckSuccess) (Stabilize)
t20:(Stabilize. [Stabilize =False])/send(StabilizeFailure)(Stabilize)
t21:(Stabilize. [Stabilize =True])/send(StabilizeSuccess) (Throttle)
t22:(Throttle. [Throttle =False])/send(ThrottleFailure)(Throttle)
t23:(Throttle. [Throttle =True])/send(Throttle Success)
t24:(Init. [startSequence=True],Launch)/ (EnviCheck)
t25:(EnviCheck [EnviCheck =False])/send(EnviCheckFailure)(EnviCheck)
t26:(EnviCheck . [EnviCheck =True])/send(EnviCheck Success) (AbortClear)
t27:(AbortClear. [AbortClear =False])/send(AbortClear Failure)(AbortClear)
t28:(AbortClear. [AbortClear =True])/send(AbortClear Success)
t29:(Init. [startSequence=True],Flight)/ (RadioRange)
t30:(RadioRange. [RadioRange =False)/send(RadioRangeFailure) (RadioRange)
t31:(RadioRange. [RadioRange =True])/send(RadioRangeSuccess)(FlightCmds)
t32:(FlightCmds. [FlightCmds =False])/send(FlightCmdsFailure) (FlightCmds)
t33:(FlightCmds. [FlightCmds =True])/send(FlightCmdsSuccess) (Idle)
t34:(End. [endSequence =True])/send(Success)

Figure 4.10: UAV CEFSM

4.4.2 UAV System Behavioral Model

There are 19 unique states and 33 transitions in the model. The test suite $CT = bt_1$ where

$bt_1 = \{s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_4, s_5, s_6, s_6, s_7, s_7, s_8, s_8, s_9, s_9, s_9, s_{10}, s_{11}, s_{11}, s_{12},$

$s_{13}, , s_{13}, s_{13}\}$, $s_{14}, s_{15}, s_{15}, s_{16}, s_{16}, s_{16}, s_{17}, s_{18}, s_{18}, s_{19}, s_{19}, s_{19}, s_{20}\}$

The paths are feasible based on reachability analysis, as there are no conflicting predicates between transitions.

4.4.3 Security Attacks on a UAV

Security attacks can be injected into any state that gets a command from the ground operator or GPS data. Any attack on a UAV has the potential to be catastrophic. Under most circumstances, it would be intended for the UAV to stay on its mission and mitigate. Therefore, most attacks on the UAV can be mitigated with a roll back and retry. There are only two cases in which it would be worth attempting to get the UAV to return home. Those would be a flight command compromised or a spoofed flight command. These are the two cases when it might be the intent of the attack to carry out malicious activity with the UAV. Attacks to the UAV are shown in Table 4.15.

Table 4.15: Possible UAV Attacks

Attack	**Type**	**Description**
A_1	DOS RFJam	Floods RF Channels
A_2	DOS Flood	Floods Comm Channels
A_3	Sniff & Spoof	Listens then injects Packets
A_4	Sniff & MITM	Listens, Forwards and Injects Packets
A_5	DOS GPS JAM	Flood GPS Channels

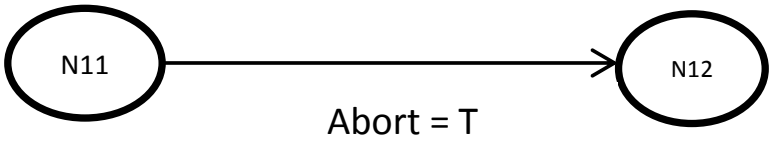

Figure 4.11: Roll Forward Mitigation Model

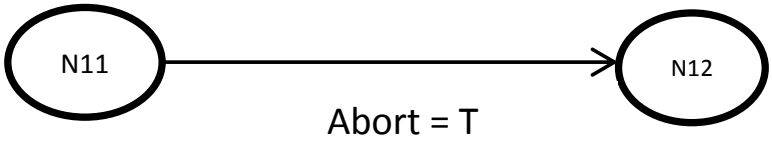

Figure 4.12: Roll Back Mitigation Model

4.4.4 UAV System Mitigation Models

Mitigations for this case study include a roll-back to a safe state or a roll-forward to a safe state (such as a go home command), as these are the only two feasible mitigations. Mitigation models are shown in Figure 4.11 and Figure 4.12.

$MT = \{mt_1..mt_2\}$

$mt_1 = (N_1, s_1)$

$mt_2 = (N_{11}, N_{12})$

Table 4.16: UAV Attack Applicability Matrix

A/CT	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	s_{13}	s_{14}	s_{15}	s_{16}	s_{17}	s_{18}	s_{19}	s_{20}
a_1	0	1	1	1	0	1	1	1	1	0	1	1	1	0	1	1	0	1	1	0
a_2	0	1	1	1	0	1	1	1	1	0	1	1	1	0	1	1	0	1	1	0
a_3	0	1	1	1	0	1	1	1	1	0	1	1	1	0	1	1	0	1	1	0
a_4	0	1	1	1	0	1	1	1	1	0	1	1	1	0	1	1	0	1	1	0
a_5	0	1	1	1	0	1	1	1	1	0	1	1	1	0	1	1	0	1	1	0

4.4.5 UAV Attack Applicability

We define I $=\sum_{i=1}^{5} len(t_i)$ state s to select for attack a. Concatenating the tests gives us the number of states in test suite $CT = bt_1 = \{s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_4, s_5, s_6, s_6, s_7, s_7, s_8, s_8, s_9, s_9, s_9, s_{10}, s_{11}, s_{11}, s_{12}, s_{13}, , s_{13}, s_{13}, s_{14}, s_{15}, s_{15}, s_{16}, s_{16}, s_{16}, s_{17}, s_{18}, s_{18}, s_{19}, s_{19}, s_{19}\}$

There are 37 positions in this model. We now apply coverage criteria for the above positions ($1 \leq s \leq 37$) and attacks ($1 \leq a \leq 5$).

Coverage Criteria 1: all states, all applicable attacks. The required (s, a) combinations are shown in Table 4.4.6 as 1 entries. This required 165 tests. This approach is practical but would not be scaleable in a larger model. Criteria 1 depends on the size of the test suite, the number of attack types, and where the attacks can occur.

Coverage Criteria 2 and 3: all unique nodes, all applicable attacks and all tests, all unique nodes, all applicable attacks. Demonstrated in Table 4.4.6 requires 70 tests.

Coverage Criteria 4: all tests, all unique nodes, some attacks. This criteria only injects attacks at one unique test point. Demonstrated in Table 4.4.6 requires 5 tests.

4.4.6 Mitigation Requirements

Mitigation requirements are summarized in Table 4.17. The tables list each state with possible attacks, the risks of the attack being executed as well as the mitigation required and the weaving rule we will use to mitigate the attack. If a state is not listed in the table, it is because no attack is possible in that state (i.e. idle states). For example in state (s_2) or (battery check compromised) an attacker could execute either a RF DOS attack (A1) or a DOS Flood attack (A2) which would result in a confusing battery result. The mitigation would be to roll the system back to the idle state (s_1) and attempt to check the battery again.

Table 4.17: Mitigation Test Suite

State	Attack	Risk if Attack Successful	MT Mitigation	WR
s1		NA		
s2	A1-A5	Batt Check	M1 Rollback s1	WR3 Opt 1
s3	A1-A5	Prop Check Compromised	M1 RollBack s1	WR3 Opt 1
s4	A1-A5	Sensor Check	M1 Rollback s1	WR3 Opt 1
s5		NA		
s6	A1-A5	Mode Check	M1 Rollback s5	WR3 Opt 1
s7	A1-A5	Home Recorded	M1 Rollback s5	WR3 Opt 1
s8	A1-A5	Prop Arm	M1 Rollback s5	WR3 Opt 1
s9	A1-A5	Arm Check	M1 Rollback s5	WR3 Opt 1
s10		NA		
s11	A1-A5	Man Control	M1 Rollback s10	WR3 Opt 1
s12	A1-A5	Stab Control	M1 Rollback s10	WR3 Opt 1
s13	A1-A5	Throttle	M1 Rollback s10	WR3 Opt 1
s14		NA		
s15	A1-A5	Enviro Check	M1 Rollback s14	WR3 Opt 1
s16	A1-A5	Abort Area	M1 Rollback s14	WR3 Opt 1
s17		NA		
s18	A1-A5	Radio	M1 Rollback s17	WR3 Opt 1
s19	A1-A5	Flight Command	M1 Rollforward	WR2 Opt 1

A/CT	bt_1																																					
	s_1	s_2	s_2	s_3	s_3	s_4	s_4	s_4	s_5	s_6	s_6	s_7	s_7	s_8	s_8	s_9	s_9	s_9	s_{10}	s_{11}	s_{11}	s_{12}	s_{12}	s_{13}	s_{13}	s_{13}	s_{14}	s_{15}	s_{15}	s_{16}	s_{16}	s_{16}	s_{17}	s_{18}	s_{18}	s_{19}	s_{19}	s_{20}
a_1	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	0
a_2	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	0
a_3	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	0
a_4	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	0
a_5	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	0

Table 4.18: UAV Attack Position Matrix- Criteria 1

A/CT	bt_1																																					
	s_1	s_2	s_2	s_3	s_3	s_4	s_4	s_4	s_5	s_6	s_6	s_7	s_7	s_8	s_8	s_9	s_9	s_9	s_{10}	s_{11}	s_{11}	s_{12}	s_{12}	s_{13}	s_{13}	s_{13}	s_{14}	s_{15}	s_{15}	s_{16}	s_{16}	s_{16}	s_{17}	s_{18}	s_{18}	s_{19}	s_{19}	s_{20}
a_1	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0
a_2	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0
a_3	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0
a_4	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0
a_5	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0

Table 4.19: UAV Attack Position Matrix- Criteria 2 and 3

A/CT	bt_1																																					
	s_1	s_2	s_2	s_3	s_3	s_4	s_4	s_4	s_5	s_6	s_6	s_7	s_7	s_8	s_8	s_9	s_9	s_9	s_{10}	s_{11}	s_{11}	s_{12}	s_{12}	s_{13}	s_{13}	s_{13}	s_{14}	s_{15}	s_{15}	s_{16}	s_{16}	s_{16}	s_{17}	s_{18}	s_{18}	s_{19}	s_{19}	s_{20}
a_1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a_2	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a_3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a_4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
a_5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

Table 4.20: UAV Attack Position Matrix- Criteria 4

Table 4.21: Security Mitigation Test for Criteria 4

Security Test	**SMT**	**bt**
smt_1	$s_1, s_2, N_1, s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_4, s_5, s_6, s_6, s_7, s_7, s_8, s_8,$ $s_9, s_9, s_9, s_{10}, s_{11}, s_{11}, s_{12},$ $s_{13}, s_{13}, s_{13}, s_{14}, s_{15}, s_{15}, s_{16},$ $s_{16}, s_{16}, s_{17}, s_{18}, s_{18}, s_{19}, s_{19}, s_{19}, s_{20}$	bt_1
smt_2	$s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_4, s_5, s_6, s_6, s_7, N_{11}, N_{12}$	bt_1
smt_3	$s_1, s_2, N_1, s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_4, s_5, s_6, s_6, s_7, s_7, s_8,$ $s_8, s_9, s_9, s_9, s_{10}, s_{11}, s_{11}, s_{12},$ s_{13}, N_{11}, N_{12}	bt_1
smt_4	$s_1, s_2, N_1, s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_4, s_5, s_6, s_6, s_7, s_7, s_8,$ $s_8, s_9, s_9, s_9, s_{10}, s_{11}, s_{11}, s_{12},$ $s_{13}, , s_{13}, s_{13}, s_{14}, s_{15}, N_{11}, N_{12}$	bt_1
smt_5	$s_1, s_2, N_1, s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_4, s_5, s_6, s_6, s_7, s_7, s_8,$ $s_8, s_9, s_9, s_9, s_{10}, s_{11}, s_{11}, s_{12},$ $s_{13}, , s_{13}, s_{13}, s_{14}, s_{15}, s_{15}, s_{16}, s_{16}, s_{16}$ $, s_{17}, s_{18}, s_{18}, s_{19}, N_{11}, N_{12}$	bt_1

4.4.7 UAV Security Test Suite

The security test suite for a UAV is defined in Table 4.21. This test suite covers the tests that would be required for criteria 4 noted above. We complete the example by demonstrating executable tests are as follows:

$SMT1$;

$s_1 : (Init, [startSequence = True], startPrearm)$

$(BatteryCheck)$

$s_2 : (BatteryCheck.[BatteryCheck = True])$

$/send(BatteryCheckSuccess)(BatteryCheck)$

$N_1 : (RollBackSafe, [AttackType = a_1][SafeState = s_1],$

$RollBackSuccess/send(RollBackSuccess)(RollBackSafe)$

$s_1 : (Init, [startSequence = True], startPrearm)$

$/(BatteryCheck)$

$s_2 : (BatteryCheck.[BatteryCheck = True])$

$/send(BatteryCheckSuccess)(BatteryCheck)$

$s_2 : (BatteryCheck.[BatteryCheck = True])$

$/send(BatteryCheckSuccess)(PropCheck)$

$s_3 : (PropCheck.[PropCheck = True])$

$/send(PropCheckSuccess)(PropCheck)$

$s_3 : (PropCheck.[PropCheck = True])$

$/send(PropCheckSuccess)(SensorCheck)$

$s_4 : (SensorCheck.[SensorCheck = True])$

$/send(SensorCheckSuccess)(SensorCheck)$

$s_4 : (SensorCheck.[SensorCheck = True])$

$/send(SensorCheckSuccess)(Init)$

$s_5 : (Init.[startSequence = True], startArming)$

$/(ModeCheck)$

$s_6 : (ModeCheck.[ModeCheck = True])$

$/send(ModeCheckSuccess)(HomePosition)$

$s_7 : (HomePosition.[HomePosition = True])$

$/send(HomePositionSuccess)(HomePosition)$

$s_7 : (HomePosition.[HomePosition = True])$

$/send(HomePositionSuccess)(PropArmed)$

$s_8 : (PropArmed.[PropArmed = True])$

$/send(PropArmedSuccess)(PropArmed)$

$s_8 : (PropArmed.[PropArmed = True])$

$/send(PropArmedSuccess)(ArmCheck)$

$s_9 : (ArmCheck.[ArmCheck = True])$

$/send(ArmCheckSuccess)(ArmCheck)$

$s_9 : (ArmCheck.[ArmCheck = True])$

$/send(ArmCheckSuccess)(Init)$

$s_{10} : (Init.[startSequence = True],$

$Preflight)/(ManCtlSurfCheck)$

$s_{11} : (ManCtlSurfCheck.[ManCtlSurfCheck = True])$

$/send(ManCtlSurfCheckSuccess)(ManCtlSurfCheck)$

$s_{11} : (ManCtlSurfCheck.[ManCtlSurfCheck = True])$

$/send(ManCtlSurfCheckSuccess)(Stabilize)$

$s_{12} : (Stabilize.[Stabilize = True])$

$/send(StabilizeSuccess)(Stabilize)$

$s_{12} : (Stabilize.[Stabilize = True])$

$/send(StabilizeSuccess)(Throttle)$

$s_{13} : (Throttle.[Throttle = True])$

$/send(ThrottleSuccess)(Throttle)$

$s_{13} : (Throttle.[Throttle = True])$

$/send(ThrottleSuccess)(Init)$

$s_{14} : (Init.[startSequence = True],$

$Launch)/(EnviCheck)$

$s_{15} : (EnviCheck.[EnviCheck = True])$

$/send(EnviCheckSuccess)(EnviCheck)$

$s_{15} : (EnviCheck.[EnviCheck = True])$

$/send(EnviCheckSuccess)(AbortClear)$

$s_{16} : (AbortClear.[AbortClear = True])$

$/send(AbortClearSuccess)(AbortClear)$

$s_{16} : (AbortClear.[AbortClear = True])$

$/send(AbortClearSuccess)(Init)$

$s_{17} : (Init.[startSequence = True],$

$Flight)/(RadioRange)$

$s_{18} : (RadioRange.[RadioRange = True])$

$/send(RadioRangeSuccess)(RadioRange)$

$s_{18} : (RadioRange.[RadioRange = True])$

$/send(RadioRangeSuccess)(FlightCmds)$

$s_{19} : (FlightCmds.[FlightCmds = True])$

$/send(FlightCmdsSuccess)(FlightCmds)$

$s_{19} : (FlightCmds.[FlightCmds = True])$

$/send(FlightCmdsSuccess)(End)$

$s_{20} : (End.[endSequence = True])/send(Success)(EndSequence);$

$SMT2 :$

$s_1 : (Init, [startSequence = True], startPrearm)$

$/(BatteryCheck)$

$s_2 : (BatteryCheck.[BatteryCheck = True])$

$/send(BatteryCheckSuccess)(BatteryCheck)$

$s_2 : (BatteryCheck.[BatteryCheck = True])$

$/send(BatteryCheckSuccess)(PropCheck)$

$s_3 : (PropCheck.[PropCheck = True])$

$/send(PropCheckSuccess)(PropCheck)$

$s_3 : (PropCheck.[PropCheck = True])$

$/send(PropCheckSuccess)(SensorCheck)$

$s_4 : (SensorCheck.[SensorCheck = True])$

$/send(SensorCheckSuccess)(SensorCheck)$

$s_4 : (SensorCheck.[SensorCheck = True])$

$/send(SensorCheckSuccess)(Init)$

$s_5 : (Init.[startSequence = True], startArming)$

$/(ModeCheck)$

$s_6 : (ModeCheck.[ModeCheck = True])$

$/send(ModeCheckSuccess)(HomePosition)$

$s_7 : (HomePosition.[HomePosition = True])$

$/send(HomePositionSuccess)(HomePosition)$

$N_{11} : (Attack, [Attack = True][AttackType = a_2]$

$RolllForward/send(AbortMessage)(RollForwardAbort)$

$N_{12} : (RollForwardAbort, [Abort = True],$

$AbortSuccess/send(AbortSuccess)(EndSequence);$

$SMT3 :$

$s_1 : (Init, [startSequence = True], startPrearm)$

$/(BatteryCheck)$

$s_2 : (BatteryCheck.[BatteryCheck = True])$

$/send(BatteryCheckSuccess)(BatteryCheck)$

$s_2 : (BatteryCheck.[BatteryCheck = True])$

$/send(BatteryCheckSuccess)(PropCheck)$

$s_3 : (PropCheck.[PropCheck = True])$

$/send(PropCheckSuccess)(PropCheck)$

$s_3 : (PropCheck.[PropCheck = True])$

$/send(PropCheckSuccess)(SensorCheck)$

$s_4 : (SensorCheck.[SensorCheck = True])$

$/send(SensorCheckSuccess)(SensorCheck)$

$s_4 : (SensorCheck.[SensorCheck = True])$

$/send(SensorCheckSuccess)(Init)$

$s_5 : (Init.[startSequence = True], startArming)$

$/(ModeCheck)$

$s_6 : (ModeCheck.[ModeCheck = True])$

$/send(ModeCheckSuccess)(HomePosition)$

$s_7 : (HomePosition.[HomePosition = True])$

$/send(HomePositionSuccess)(HomePosition)$

$s_7 : (HomePosition.[HomePosition = True])$

$/send(HomePositionSuccess)(PropArmed)$

$s_8 : (PropArmed.[PropArmed = True])$

$/send(PropArmedSuccess)(PropArmed)$

$s_8 : (PropArmed.[PropArmed = True])$

$/send(PropArmedSuccess)(ArmCheck)$

$s_9 : (ArmCheck.[ArmCheck = True])$

$/send(ArmCheckSuccess)(ArmCheck)$

$s_9 : (ArmCheck.[ArmCheck = True])$

$/send(ArmCheckSuccess)(Init)$

$s_{10} : (Init.[startSequence = True],$

$Preflight)/(ManCtlSurfCheck)$

$s_{11} : (ManCtlSurfCheck.[ManCtlSurfCheck = True])$

$/send(ManCtlSurfCheckSuccess)(ManCtlSurfCheck)$

$s_{11} : (ManCtlSurfCheck.[ManCtlSurfCheck = True])$

$/send(ManCtlSurfCheckSuccess)(Stabilize)$

$s_{12} : (Stabilize.[Stabilize = True])$

$/send(StabilizeSuccess)(Stabilize)$

$s_{12} : (Stabilize.[Stabilize = True])$

$/send(StabilizeSuccess)(Throttle)$

$s_{13} : (Throttle.[Throttle = True])$

$/send(ThrottleSuccess)(Throttle)$

$N_{11} : (Attack, [Attack = True][AttackType = a_3]$

$RolllForward/send(AbortMessage)(RollForwardAbort)$

$N_{12} : (RollForwardAbort, [Abort = True],$

$AbortSuccess/send(AbortSuccess)(EndSequence);$

$SMT4 :$

$s_1 : (Init, [startSequence = True], startPrearm)$

$/(BatteryCheck)$

$s_2 : (BatteryCheck.[BatteryCheck = True])$

$/send(BatteryCheckSuccess)(BatteryCheck)$

$s_2 : (BatteryCheck.[BatteryCheck = True])$

$/send(BatteryCheckSuccess)(PropCheck)$

$s_3 : (PropCheck.[PropCheck = True])$

$/send(PropCheckSuccess)(PropCheck)$

$s_3 : (PropCheck.[PropCheck = True])$

$/send(PropCheckSuccess)(SensorCheck)$

$s_4 : (SensorCheck.[SensorCheck = True])$

$/send(SensorCheckSuccess)(SensorCheck)$

$s_4 : (SensorCheck.[SensorCheck = True])$

$/send(SensorCheckSuccess)(Init)$

$s_5 : (Init.[startSequence = True], startArming)$

$/(ModeCheck)$

$s_6 : (ModeCheck.[ModeCheck = True])$

$/send(ModeCheckSuccess)(HomePosition)$

$s_7 : (HomePosition.[HomePosition = True])$

$/send(HomePositionSuccess)(HomePosition)$

$s_7 : (HomePosition.[HomePosition = True])$

$/send(HomePositionSuccess)(PropArmed)$

$s_8 : (PropArmed.[PropArmed = True])$

$/send(PropArmedSuccess)(PropArmed)$

$s_8 : (PropArmed.[PropArmed = True])$

$/send(PropArmedSuccess)(ArmCheck)$

$s_9 : (ArmCheck.[ArmCheck = True])$

$/send(ArmCheckSuccess)(ArmCheck)$

$s_9 : (ArmCheck.[ArmCheck = True])$

$/send(ArmCheckSuccess)(Init)$

$s_{10} : (Init.[startSequence = True],$

$Preflight)/(ManCtlSurfCheck)$

$s_{11} : (ManCtlSurfCheck.[ManCtlSurfCheck = True])$

$/send(ManCtlSurfCheckSuccess)(ManCtlSurfCheck)$

$s_{11} : (ManCtlSurfCheck.[ManCtlSurfCheck = True])$

$/send(ManCtlSurfCheckSuccess)(Stabilize)$

$s_{12} : (Stabilize.[Stabilize = True])$

$/send(StabilizeSuccess)(Stabilize)$

$s_{12} : (Stabilize.[Stabilize = True])$

$/send(StabilizeSuccess)(Throttle)$

$s_{13} : (Throttle.[Throttle = True])$

$/send(ThrottleSuccess)(Throttle)$

$s_{13} : (Throttle.[Throttle = True])$

$/send(ThrottleSuccess)(Init)$

$s_{14} : (Init.[startSequence = True],$

$Launch)/(EnviCheck)$

$s_{15} : (EnviCheck.[EnviCheck = True])$

$/send(EnviCheckSuccess)(EnviCheck)$

$N_{11} : (Attack, [Attack = True][AttackType = a_4]$

$RolllForward/send(AbortMessage)(RollForwardAbort)$

$N_{12} : (RollForwardAbort, [Abort = True],$

$AbortSuccess/send(AbortSuccess)(EndSequence);$

$SMT5 :$

$s_1 : (Init, [startSequence = True], startPrearm)$

$/(BatteryCheck)$

$s_2 : (BatteryCheck.[BatteryCheck = True])$

$/send(BatteryCheckSuccess)(BatteryCheck)$

$s_2 : (BatteryCheck.[BatteryCheck = True])$

$/send(BatteryCheckSuccess)(PropCheck)$

$s_3 : (PropCheck.[PropCheck = True])$

$/send(PropCheckSuccess)(PropCheck)$

$s_3 : (PropCheck.[PropCheck = True])$

$/send(PropCheckSuccess)(SensorCheck)$

$s_4 : (SensorCheck.[SensorCheck = True])$

$/send(SensorCheckSuccess)(SensorCheck)$

$s_4 : (SensorCheck.[SensorCheck = True])$

$/send(SensorCheckSuccess)(Init)$

$s_5 : (Init.[startSequence = True], startArming)$

$/(ModeCheck)$

$s_6 : (ModeCheck.[ModeCheck = True])$

$/send(ModeCheckSuccess)(HomePosition)$

$s_7 : (HomePosition.[HomePosition = True])$

$/send(HomePositionSuccess)(HomePosition)$

$s_7 : (HomePosition.[HomePosition = True])$

$/send(HomePositionSuccess)(PropArmed)$

$s_8 : (PropArmed.[PropArmed = True])$

$/send(PropArmedSuccess)(PropArmed)$

$s_8 : (PropArmed.[PropArmed = True])$

$/send(PropArmedSuccess)(ArmCheck)$

$s_9 : (ArmCheck.[ArmCheck = True])$

$/send(ArmCheckSuccess)(ArmCheck)$

$s_9 : (ArmCheck.[ArmCheck = True])$

$/send(ArmCheckSuccess)(Init)$

$s_{10} : (Init.[startSequence = True],$

$Preflight)/(ManCtlSurfCheck)$

$s_{11} : (ManCtlSurfCheck.[ManCtlSurfCheck = True])$

$/send(ManCtlSurfCheckSuccess)(ManCtlSurfCheck)$

$s_{11} : (ManCtlSurfCheck.[ManCtlSurfCheck = True])$

$/send(ManCtlSurfCheckSuccess)(Stabilize)$

$s_{12} : (Stabilize.[Stabilize = True])$

$/send(StabilizeSuccess)(Stabilize)$

$s_{12} : (Stabilize.[Stabilize = True])$

$/send(StabilizeSuccess)(Throttle)$

$s_{13} : (Throttle.[Throttle = True])$

$/send(ThrottleSuccess)(Throttle)$

$s_{13} : (Throttle.[Throttle = True])$

$/send(ThrottleSuccess)(Init)$

$s_{14} : (Init.[startSequence = True],$

$Launch)/(EnviCheck)$

$s_{15} : (EnviCheck.[EnviCheck = True])$

$/send(EnviCheckSuccess)(EnviCheck)$

$s_{15} : (EnviCheck.[EnviCheck = True])$

$/send(EnviCheckSuccess)(AbortClear)$

$s_{16} : (AbortClear.[AbortClear = True])$

$/send(AbortClearSuccess)(AbortClear)$

$s_{16} : (AbortClear.[AbortClear = True])$

$/send(AbortClearSuccess)(Init)$

$s_{17} : (Init.[startSequence = True],$

$Flight)/(RadioRange)$

$s_{18} : (RadioRange.[RadioRange = True])$

$/send(RadioRangeSuccess)(RadioRange)$

$s_{18} : (RadioRange.[RadioRange = True])$

$/send(RadioRangeSuccess)(FlightCmds)$

$s_{19} : (FlightCmds.[FlightCmds = True])$

$/send(FlightCmdsSuccess)(FlightCmds)$

$N_{11} : (Attack, [Attack = True][AttackType = a_5]$

$RolllForward/send(AbortMessage)(RollForwardAbort)$

$N_{12} : (RollForwardAbort, [Abort = True],$

$AbortSuccess/send(AbortSuccess)(EndSequence);$

4.4.8 Discussion

We have demonstrated that our method can be applied to a UAV model. The goal of this case study was to add a new attack and apply it to a semi-autonomous air vehicle with a ground operator. We show that we can model required mitigations for security attacks and generate mitigation tests. We can identify criteria for covering attack scenarios in a systematic way. We can also use behavioral tests, mitigation test and attack scenarios to build a systematic approach and build our security test suite (MBST). It is a scalable approach that can be applied to a UAV.

4.5 Case Study 4: Robot

In this section we demonstrate our approach with a robot example to show how to test for security attacks. The goal of this case study is to demonstrate our approach on a critical system. Robots are used to for search and rescue missions, explosive containment and other critical functions. They rely on public communication channels and therefore are vulnerable to security attacks.

A robot consists of a semi-autonomous vehicle and a ground operator. The semi-autonomous vehicle is comprised of a camera and a GPS device. Robots can vary in complexity, for this example we focus on a very simple example. The controller is responsible for powering up the robot, retrieving the robots location, sending the robot a new location and starting and stopping the camera. The robot simply executes the commands it receives from the controller. For this example, we will always attempt to retry the commands instead of aborting.

Figures 4.13 and 4.14 show the CEFSM model of the robot system including states, transitions, variables, events, and messages. The CEFSM begins with the initialization phase. In this phase, the model transitions with t_1 from an "idle" state

s_1 (where no attacks are possible) to the "power on" state s_2. If the "power on" s_2 state fails, retry's are attempted with t_2. if the "power on" state s_2 is successful, a success message is sent and the model transitions with t_3 to the "location check" state s_3. The "location check" state s_3 is retried with a t_4 transition on a failure. For a success, the "location check" state s_3 sends a success message and transitions to a "camera on" state s_4. The model transitions with t_7 to a move state (s_6). If successful, the model transitions with t_9 to a "camera off" state. If successful, the robot transitions to the "end" state (s_8).

4.5.1 Security Attacks on a Robot

Security attacks can be injected at each state where there is a communication with the controller. Attacks are the most critical when the robot is being commanded to move as the robot could be manipulated to be used for malice. The attacks that are possible in a robot system are defined in Table 4.22.

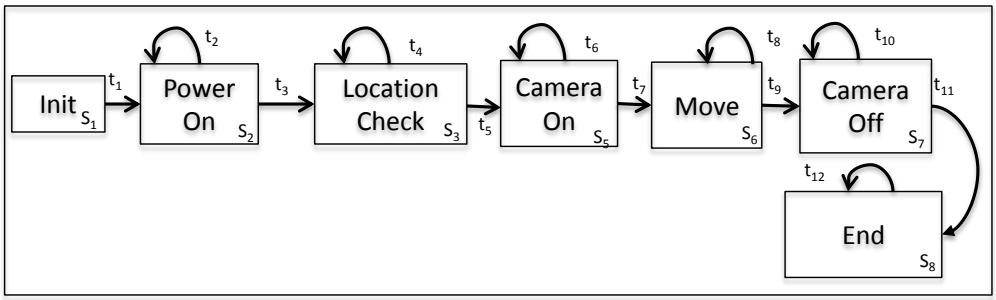

Figure 4.13: Robot CEFSM Part 1

Table 4.22: Possible Robot Attacks

Attack	**Type**	**Description**
a_1	DOS RFJam	Floods RF Channels
a_2	DOS Flood	Floods Comm Channels
a_3	Sniff & Spoof	Listens then injects Packets
a_4	Sniff & MITM	Listens, Forwards and Injects Packets

Table 4.23: Robot Attacks

State	Attack	Risk	MT Mitigation	WR
s_1		NA		
s_2	a_1-a_4	confusion	M3 M1/M4 Rollback to s1	WR3 Option 1
s_3	a_1-a_4	confusion	M3 M1/M4 Rollback to s1	WR3 Option 1
s_4	a_1-a_4	confusion	M3 M1/M4 Rollback to s1	WR3 Option 1
s_5	a_1-a_4	confusion	M3 M1/M4 Rollback to s1	WR3 Option 1
s_6	a_1-a_4	confusion	M3 M1/M4 Rollback to s1	WR3 Option 1
s_7	a_1-a_4	confusion	M3 M1/M4 Rollback to s1	WR3 Option 1
s_8		NA		

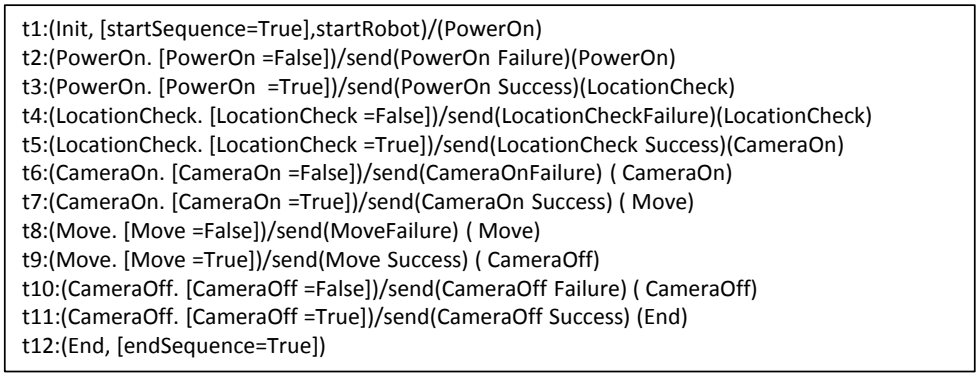

Figure 4.14: Robot CEFSM Part 2

4.5.2 Behavioral Model(BM),Test Criteria (BC), and Test Suite (BT)

Figure 4.13 depicts the behavioral model of a robot in Communicating Extended Finite State Machine (CEFSM) format. The model contains 8 unique states and 12 transitions. Test paths are determined by following both states and transition in the CEFSM. The specifics of transitions can be found in [37]. The test suite

$CT = \{bt_1\}$ where

$bt_1 = \{\ s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_5, s_5, s_6, s_6, s_7, s_7, s_8\}$

Assuming edge coverage is required, the test paths BT fulfill this requirement. By using reachability analysis, we find that these paths are feasible because there are no conflicting predicates along the paths. Some attacks are only feasible in certain states. Table 4.24 shows the attack applicability matrix AM. We choose (p,a) pairs that meet node attack coverage criteria. Note that in this case study, A5 is not applicable as there is no GPS system on this robot example.

Table 4.24: Robot Attack Applicability Matrix

A/CT	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8
a_1	0	1	1	1	1	1	1	0
a_2	0	1	1	1	1	1	1	0
a_3	0	1	1	1	1	1	1	0
a_4	0	1	1	1	1	1	1	0

4.5.3 Security Test Requirements

Next, we determine security test requirements (p, a) for coverage criteria 1-4 (SR_1-SR_4). As per the definition of CT CT is given as CT = bt_1 = { bt_1 = { s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_5, s_5, s_6, s_6, s_7, s_7, s_8} There are 14 positions. We now apply coverage criteria for the above positions ($1 \leq s \leq 14$) and attacks ($1 \leq a \leq 4$).

Coverage Criteria 1: all states, all applicable attacks. The required (p,a) combinations are shown in Table 4.25 as 1 entries. This requires 52 security tests. While theoretically feasible, it is a large number of tests. This is not only due to the length of CT, but also due to the large proportion of 1 entries in AM, i.e. whether attacks are feasible in most states or not.

Coverage Criteria 2 and 3: all unique nodes, all applicable attacks and all tests, all unique nodes, all applicable attacks. The robot example is very sequential. Therefore, criteria 2 and criteria 3 have the same matrix, as shown in Table 4.26. They both require 28 security tests.

Coverage Criteria 4: all tests, all unique nodes, some attacks. This criteria does not require that all attacks be injected at every state even though each attack must be selected at least once. (p,a) combinations that meet this requirement are shown in Table 4.27 as 1 entries. There are 4 security tests required.

Table 4.25: Robot Security Test Requirements - Criteria 1

	bt_1														
A/CT	s_1	s_2	s_2	s_3	s_3	s_4	s_4	s_5	s_5	s_6	s_6	s_7	s_7	s_8	s_8
a_1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0
a_2	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0
a_3	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0
a_4	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0

Table 4.26: Robot Security Test Requirements - Criteria 2 and 3

	bt_1														
A/CT	s_1	s_2	s_2	s_3	s_3	s_4	s_4	s_5	s_5	s_6	s_6	s_7	s_7	s_8	s_8
a_1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
a_2	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
a_3	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
a_4	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0

4.5.4 Mitigation Requirements, Models, Security Mitigation Tests

The mitigation requirements are summarized in Table 4.23. The tables list each state with possible attacks, the risks of the attack being executed as well as the mitigation required and the weaving rule we will use to mitigate the attack. If a state is not listed in the table, it is because no attack is possible in that state (i.e. idle or end states). For example in s_1 or s_8, no attacks are possible.

Table 4.27: Robot Security Test Requirements - Criteria 4

	bt_1														
A/CT	s_1	s_2	s_2	s_3	s_3	s_4	s_4	s_5	s_5	s_6	s_6	s_7	s_7	s_8	s_8
a_1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
a_2	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
a_3	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
a_4	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0

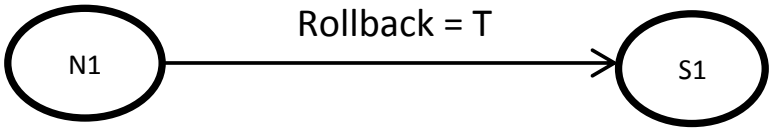

Figure 4.15: Roll Back Mitigation Model M1

Table 4.28: Security Mitigation Test for Criteria 4

Security Test	**SMT**	**bt**
smt_1	$s_1, s_2, N_1, s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_5, s_5, s_6, s_6,$ s_7, s_7, s_8, s_8	bt_1
smt_2	$s_1, s_2, s_2, s_3, s_3, N_1, s_1, s_2, s_2, s_3,$ $s_3, s_4, s_4, s_5, s_5, s_6, s_6, s_7, s_7, s_8, s_8$	bt_1
smt_3	$s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_5, N_1,$ $s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_5, s_6, s_6,$ s_7, s_7, s_8, s_8	bt_1
smt_4	$s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_5, s_5, s_6, s_6,$ $N_1, s_1, s_2, s_2, s_3, s_3, s_4, s_4, s_5, s_5, s_6,$ s_6, s_7, s_7, s_8, s_8	bt_1

4.5.5 Robot System Mitigation Models

Security attacks can be injected into any state that gets a command from the controller. Mitigations for this test case include a roll-back to a safe state, this is the only feasible mitigation. The mitigation model is shown in Figure 4.15.

4.5.6 Robot System Security Test Suite

The security test suite for a robot is defined in Table 4.28. This test suite covers the tests that would be required for criteria 4 noted above. We complete the example by demonstrating executable tests are as follows:

$SMT1:$

$s_1 : (Init, [startSequence = True], startRobot)$

$/(PowerOn)$

$s_2 : (PowerOn.[PowerOn = True])/send(PowerOnSuccess)$

$(PowerOn)$

$N_1 : (RollBackSafe, [AttackType = a_1][SafeState = s_1],$

$RollBackSuccess/send(RollBackSuccess)(RollBackSafe)$

$s_1 : (Init, [startSequence = True], startRobot)$

$/(PowerOn)$

$s_2 : (PowerOn.[PowerOn = True])/send(PowerOnSuccess)$

$(PowerOn)$

$s_2 : (PowerOn.[PowerOn = True])/send(PowerOnSuccess)$

$(LocationCheck)$

$s_3 : (LocationCheck.[LocationCheck = True])/send(LocationCheckSuccess)$

$(LocationCheck)$

$s_3 : (LocationCheck.[LocationCheck = True])/send(LocationCheckSuccess)$

$(CameraOn)$

$s_5 : (CameraOn.[CameraOn = True])/send(CameraOnSuccess)$

$(CameraOn)$

$s_5 : (CameraOn.[CameraOn = True])/send(CameraOnSuccess)$

$(Move)$

$s_6 : (Move.[Move = True])/send(MoveSuccess)$

$(Move)$

$s_6 : (Move.[Move = True])/send(MoveSuccess)$

$(CameraOff)$

$s_7 : (CameraOff.[CameraOff = True])/send(CameraOffSuccess)$

$(CameraOff)$

$s_7 : (CameraOff.[CameraOff = True])/send(CameraOffSuccess)$

(End)

$s_8 : (End, [endSequence = True])(EndSequence);$

$SMT2 :$

$s_1 : (Init, [startSequence = True], startRobot)$

$/(PowerOn)$

$s_2 : (PowerOn.[PowerOn = True])/send(PowerOnSuccess)$

$(PowerOn)$

$s_2 : (PowerOn.[PowerOn = True])/send(PowerOnSuccess)$

$(LocationCheck)$

$s_3 : (LocationCheck.[LocationCheck = True])/send(LocationCheckSuccess)$

$(LocationCheck)$

$s_3 : (LocationCheck.[LocationCheck = True])/send(LocationCheckSuccess)$

$(CameraOn)$

$N_1 : (RollBackSafe, [AttackType = a_2][SafeState = s_1],$

$RollBackSuccess/send(RollBackSuccess)(RollBackSafe)$

$s_1 : (Init, [startSequence = True], startRobot)$

$/(PowerOn)$

$s_2 : (PowerOn.[PowerOn = True])/send(PowerOnSuccess)$

$(PowerOn)$

$s_2 : (PowerOn.[PowerOn = True])/send(PowerOnSuccess)$

$(LocationCheck)$

$s_3 : (LocationCheck.[LocationCheck = True])/send(LocationCheckSuccess)$

$(LocationCheck)$

$s_3 : (LocationCheck.[LocationCheck = True])/send(LocationCheckSuccess)$

$(CameraOn)$

$s_5 : (CameraOn.[CameraOn = True])/send(CameraOnSuccess)$

$(CameraOn)$

$s_5 : (CameraOn.[CameraOn = True])/send(CameraOnSuccess)$

$(Move)$

$s_6 : (Move.[Move = True])/send(MoveSuccess)$

$(Move)$

$s_6 : (Move.[Move = True])/send(MoveSuccess)$

$(CameraOff)$

$s_7 : (CameraOff.[CameraOff = True])/send(CameraOffSuccess)$

$(CameraOff)$

$s_7 : (CameraOff.[CameraOff = True])/send(CameraOffSuccess)$

(End)

$s_8 : (End, [endSequence = True])(EndSequence);$

$SMT3 :$

$s_1 : (Init, [startSequence = True], startRobot)$

$/(PowerOn)$

$s_2 : (PowerOn.[PowerOn = True])/send(PowerOnSuccess)$

$(PowerOn)$

$s_2 : (PowerOn.[PowerOn = True])/send(PowerOnSuccess)$

$(LocationCheck)$

$s_3 : (LocationCheck.[LocationCheck = True])/send(LocationCheckSuccess)$

$(LocationCheck)$

$s_3 : (LocationCheck.[LocationCheck = True])/send(LocationCheckSuccess)$

$(CameraOn)$

$s_5 : (CameraOn.[CameraOn = True])/send(CameraOnSuccess)$

$(CameraOn)$

$N_1 : (RollBackSafe, [AttackType = a_1][SafeState = s_1],$

$RollBackSuccess/send(RollBackSuccess)(RollBackSafe)$

$s_1 : (Init, [startSequence = True], startRobot)$

$/(PowerOn)$

$s_2 : (PowerOn.[PowerOn = True])/send(PowerOnSuccess)$

$(PowerOn)$

$s_2 : (PowerOn.[PowerOn = True])/send(PowerOnSuccess)$

$(LocationCheck)$

$s_3 : (LocationCheck.[LocationCheck = True])/send(LocationCheckSuccess)$

$(LocationCheck)$

$s_3 : (LocationCheck.[LocationCheck = True])/send(LocationCheckSuccess)$

$(CameraOn)$

$s_5 : (CameraOn.[CameraOn = True])/send(CameraOnSuccess)$

$(CameraOn)$

$s_5 : (CameraOn.[CameraOn = True])/send(CameraOnSuccess)$

$(Move)$

$s_6 : (Move.[Move = True])/send(MoveSuccess)$

$(Move)$

$s_6 : (Move.[Move = True])/send(MoveSuccess)$

$(CameraOff)$

$s_7 : (CameraOff.[CameraOff = True])/send(CameraOffSuccess)$

$(CameraOff)$

$s_7 : (CameraOff.[CameraOff = True])/send(CameraOffSuccess)$

(End)

$s_8 : (End, [endSequence = True])(EndSequence);$

$SMT4 :$

$s_1 : (Init, [startSequence = True], startRobot)$

$/(PowerOn)$

$s_2 : (PowerOn.[PowerOn = True])/send(PowerOnSuccess)$

$(PowerOn)$

$s_2 : (PowerOn.[PowerOn = True])/send(PowerOnSuccess)$

$(LocationCheck)$

$s_3 : (LocationCheck.[LocationCheck = True])/send(LocationCheckSuccess)$

$(LocationCheck)$

$s_3 : (LocationCheck.[LocationCheck = True])/send(LocationCheckSuccess)$

$(CameraOn)$

$s_5 : (CameraOn.[CameraOn = True])/send(CameraOnSuccess)$

$(CameraOn)$

$s_5 : (CameraOn.[CameraOn = True])/send(CameraOnSuccess)$

$(Move)$

$s_6 : (Move.[Move = True])/send(MoveSuccess)$

$(Move)$

$s_6 : (Move.[Move = True])/send(MoveSuccess)$

$(CameraOff)$

$N_1 : (RollBackSafe, [AttackType = a_1][SafeState = s_1],$

$RollBackSuccess/send(RollBackSuccess)(RollBackSafe)$

$s_1 : (Init, [startSequence = True], startRobot)$

$/(PowerOn)$

$s_2 : (PowerOn.[PowerOn = True])/send(PowerOnSuccess)$

$(PowerOn)$

$s_2 : (PowerOn.[PowerOn = True])/send(PowerOnSuccess)$

$(LocationCheck)$

$s_3 : (LocationCheck.[LocationCheck = True])/send(LocationCheckSuccess)$

$(LocationCheck)$

$s_3 : (LocationCheck.[LocationCheck = True])/send(LocationCheckSuccess)$

$(CameraOn)$

$s_5 : (CameraOn.[CameraOn = True])/send(CameraOnSuccess)$

$(CameraOn)$

$s_5 : (CameraOn.[CameraOn = True])/send(CameraOnSuccess)$

$(Move)$

$s_6 : (Move.[Move = True])/send(MoveSuccess)$

$(Move)$

$s_6 : (Move.[Move = True])/send(MoveSuccess)$

$(CameraOff)$

$s_7 : (CameraOff.[CameraOff = True])/send(CameraOffSuccess)$

$(CameraOff)$

$s_7 : (CameraOff.[CameraOff = True])/send(CameraOffSuccess)$

$s_8 : (End, [endSequence = True])(EndSequence);$

4.5.7 Discussion

We have demonstrated that our approach can be applied to a simple robot. The purpose of this case study is to apply our method to a different type of vehicle. We will also copare this method to other published methods from our background chapter. This approach has limited actions, in that it always rolls back to the safe state and retries. This was selected as aborting the mission would generally abandon the robot in an unknown area.

4.6 Case Study Evaluation

4.6.1 Applicability

Using our case studies, we show applicability to various systems. Our approach was used to generate a security test suite to identify and mitigate security attacks in a launch vehicle (both pre and post launch) and a UAV. Our approach was applied to four examples and we could achieve edge coverage with a small test suite.

In the He method the trust boundary line is drawn between the areas that attacks can occur. The functions above the trust boundary line are considered to not be high risk interfaces. Functions below the trust boundary are considered to be high risk. As each interface needs to be tested with each attack, the state space becomes very large, very quickly. In the example, a very simple example with 5 interfaces and 5 attacks requires 25 tests. In our test cases in section 4, we have between 24 and 34 state transitions which would result in 120 and 170 tests.

We can compare our robot model to the He [41] method from the Background Chapter. Modeling the robot with 5 interfaces and 5 attacks would result in 25 tests. We demonstrate using the CEFSM that testing every interface with every attack would result in 52 tests (maximum amount of tests). We also demonstrate that if we

used criteria 2 or criteria 3, we would need 28 tests. However, if we used criteria 4, we would only require 4 tests. Which is a significant improvement over they He method.

4.6.2 Generalizability

In our case studies, we have used CEFSM models for each of the systems. However we could use different model types such as UML to demonstrate generalization. To demonstrate generalization, we use UML models to describe the behavior of an Activmedia Pioneer 3 AT (All Terrain) robot. Our robot contains a wireless network adapter. The wireless interface operates on a standard 2.4 GHz frequency band. The robot is capable of being commanded over a wireless network with ARIA (Advanced Robot Interface for Applications) software. The UML activity diagram is shown in Figure 4.16.

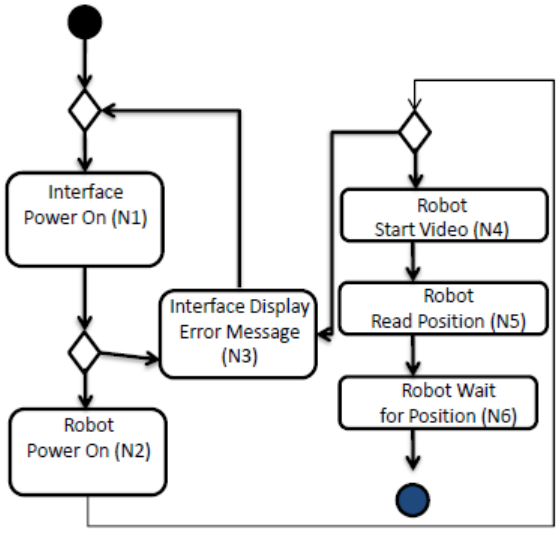

Figure 4.16: Robot UML

4.6.3 Effectiveness

We prove effectiveness in detecting and mitigating attacks by determining if security flaws would be found. We demonstrated this in a lab environment, with a commercial off the shelf (COTS) Activmedia Pioneer 3 AT robot. We commanded the robot to follow a yellow line on the floor. We then set up a malicious computer, which was intended to confuse the robot's wireless communication transmission. Our malicious computer successfully sniffed data, decoded WEP, and viewed data on wireless bus using Wireshark free software then sniffed data was used to spoof messages to the robot using LibNet free software. Our protocol successfully highlighted the areas of the black box robot that were vulnerable for attack.

4.6.4 Efficiency

We can exhibit efficiency in the size of our test suites by examining the results. We have demonstrated four case studies with varying sizes of models. They all resulted in a small test suite that gave adequate coverage to verify they can mitigate against a security attack. Table 4.29 compares the number of states and transitions in each of the case studies.

Table 4.29: Method Applicability Matrix

	Prelaunch	Postlaunch	UAV	Robot
States	21	14	19	8
Transitions	34	24	33	12
Attacks	4	4	5	4
Criteria 1	126	92	165	53
Criteria 2 and 3	52	40	70	28
Criteria 4	5	4	5	5

Chapter 5: Future Work

5.1 Attack and Mitigation Simulator

In the background chapter, we mentioned the method by Beech *et al.* [14] that designed, implemented and evaluated an attack simulator tool. We could expand on this tool to generate a simulation and mitigation tool. Their tool would need to be expanded to:

- A compiled program P

- Protection Mechanism

- Input to P

- Mitigation to attack M

The current attack simulator can insert attacks at different points and evaluate how the security mechanism reacts. The simulator injects machine instructions as attacks. It could be expanded to inject machine instructions that mitigate the attack as well. This method would be valuable, because the integrity of the compiled program (or black box) would not be altered. The current simulator is built using a dynamic compiler. The dynamic compiler goes into a loop where it builds machine instructions that would execute sequentially. At the same time the dynamic compiler could generate mitigation machine instructions.

Building this simulator would be a valuable addition to our approach as it would make the attacks, behavior and mitigations executable. It would also give us an environment to compare the models against an actual execution.

5.2 Different Models

Additional work could be done to model the autonomous vehicles with different modeling tools such as UML, Petri Nets or SDL. Using different models would validate the ability of the approach independent of the models. A comparison could be done to determine which models are most efficient, easiest to use and provide the most information. All models would have to be able to model concurrent processes.

Chapter 6: Conclusions

This dissertation proposes a black box scalable MBT approach for generating a security test suite for autonomous and semi-autonomous vehicles. We have demonstrated that we can use graph-based modeling (such as CEFSM) to model the behavior of a system, an attack as well as a mitigation. We have also demonstrated that we can keep the models smaller by not integrating all attacks and mitigations at all attack points. Rather, we weave in only the necessary mitigation test paths.

In the first step of our process, we demonstrate that we can create a black box behavioral model (BM) using graph-based testing criteria of the system. Our BM accurately depicts the intended behavior of the system. With this model, we show that we can associate coverage criteria and build a behavioral test suite (BT). It is important for us to identify which attacks are possible at given states. Attacks can be divided into 5 basic categorizations. However, we can combine any of the attacks to create new attacks. We understand that attackers are constantly evolving and that in the future, more categories could be required. Our approach allows for new attacks to easily be appended to the attack list. The second step of our process, we have shown that we can determine attack types, an attack applicability matrix and mitigation requirements. We have built mitigation models (MM) for each type of attack. We have also shown that we can use coverage criteria (MC) to determine the test paths for each model (MT). We have also demonstrated that not all attacks are applicable in all states and we have defined the attack applicability matrix (AM). In the third

step of our process, we have demonstrated that we can determine at which point in the execution an attack could occur. This information is used with coverage criteria to derive our security test requirements. In our final step we have demonstrated that we can define weaving rules to determine which mitigation will be woven into the original test paths at attack points.

We have also demonstrated that while it might seem like a good idea to integrate all attack and mitigation models at all attack points, it would quickly make the model too large. We have shown that as long as the models are created using a graph based model tool, they can be woven together to mitigate attacks in test paths.

In this dissertation, we have conducted three case studies using variations of autonomous vehicles with different model sizes, possible attacks and mitigations. In our pre-launch test case, we demonstrated that there were multiple rendezvous points in which an attacker could compromise the vehicle, the results of such a compromise would be devastating and could result in a loss of the vehicle or loss of life. We discussed that early in the launch, the system could be rolled back and states retried if an attack were to happen. However, once the fueling begins on the launch vehicle a mitigation may only be to abort the launch to prevent explosion.

In our post launch test case, we showed that the time from launch to orbit is relatively short, an attacker could compromise the launch vehicle and use it for malicious purposes. In the case study, we noted that during a DOS attack, we could potentially roll the system back to a safe state and retry. However, spoofed commands that could prematurely separate stages of the vehicle would need to be mitigated by rolling forward to an abort state, which would ignite the SRM and destroy the vehicle and the payload.

As described in our UAV example, we have demonstrated that given known vulnerabilities in the GPS guidance system, an attacker could confuse our navigation

system and take over our vehicle. We describe that before the vehicle takes off it is important to save the home location and if the vehicle is compromised, we can roll forward to a state that forces the vehicle to fly home. In our launch vehicle examples, there is never a fear of the vehicle getting into enemy hands. That is not true with our UAV example. So we must keep it from getting confused and landing in enemy territory.

In each of the three case studies, we have successfully generated behavioral models, identified attacks that are possible, built an attack applicability matrix, identified mitigations and built a security test suite. Each of the models is a different size. However, our approach is scalable for all three. We have also successfully demonstrated that our attack is applicable to multiple types of autonomous or semi-autonomous vehicles and different stages.

Our method is a robust novel approach to security test autonomous vehicles. It is a much simpler than generating attack trees for each of the test cases and we do not have state based explosion problems. Our method is also unique because we can:

- Leverage an MBT approach to build a security test suite
- Model security attacks as well as mitigation tests
- Systematically identify criteria for attack coverage
- Use behavioral tests, mitigation test and attack scenarios to build an approach for security testing (MBST) In addition,
- We have shown the approach is scalable
- We have demonstrated that the approach will work for single or multiple attacks

- We have also shown the approach is robust enough to be applied to various types of autonomous vehicles and launch systems, such as launch vehicles or UAVs

We can expand on this work to generate an attack and mitigation simulator tool. The tool could be used to build an environment to compare models against an execution. It could also be used to try out new single or multiple attacks and evaluate the results.

Bibliography

[1] Ariane 5 flight 501 failure report by the inquiry board. *https://www.ima.umn.edu/∼arnold/disasters/ariane5rep.html*, 1996.

[2] Cryosat mission lost due to launch failure. *http://www.esa.int/Our Activities/Observing the Earth/CryoSat*, 2005.

[3] *An Efficient Event Based Approach for Verification of UML Statechart Model for Reactive Systems*, 2008.

[4] Introducing cryosat. *http://www.esa.int/Our Activities/Observing the Earth/CryoSat/Introducing CryoSat*, 2010.

[5] A. Alvi and M. Zulkernine. A natural classification scheme for software security patterns. In *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*, pages 113–120, 2011.

[6] A. Alvi and M. Zulkernine. A comparative study of software security pattern classifications. In *Availability, Reliability and Security (ARES), 2012 Seventh International Conference on*, pages 582–589, 2012.

[7] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[8] A. Andrews, S. Elakeili, and S. Boukhris. Fail-safe test generation in safety critical systems. In *Proceedings of the 2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*, HASE '14, pages 49–56, Washington, DC, USA, 2014. IEEE Computer Society.

[9] K. Androutsopoulos, D. Clark, M. Harman, Z. Li, and L. Tratt. Control dependence for extended finite state machines. *FASE '09 Proceedings Of The 12th International Conference on Fundamental Approaches to Software Engineering*, 2009.

[10] G. Antoniol. Keynote paper: Search based software testing for software security: Breaking code to make it safer. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on*, pages 87–100, 2009.

[11] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, January 2004.

[12] S. Barnum and A. Sethi. Attack patterns as a knowledge resource for building secure software. In *OMG Software Assurance Workshop: Cigital*, 2007.

[13] D. Basin, J. Doser, and T. Lodderstedt. Model driven security: From uml models to access control infrastructures. *ACM Trans. Softw. Eng. Methodol.*, 15:39–91, 2006.

[14] B. Beech, M. Tegtmeyer, and L. Pollock. An attack simulator for systematically testing program-based security mechanisms. In *17th International Symposium on Software Reliability Engineering (ISSRE'06)*, 2006.

[15] S. Bennett, S. McRobb, and R. Farmer. *Object-oriented systems analysis and design using UML*. McGraw Hill Higher Education, 2005.

[16] A. Bertolino, E. Marchetti, and A. Polini. Integration of components to test software components. *Electronic Notes in Theoretical Computer Science*, 82(6):44–54, 2003.

[17] C. Bourhfir, E. Aboulhamid, R. Dssouli, and N. Rico. A Test Case Generation Approach for Conformance Testing of SDL Systems. *Comp. Commun.*, 24(3-4):319–333, 2001.

[18] C. Bourhfir, R. Dssouli, E.M. Aboulhamid, and N. Rico. A Guided Incremental Test Case Generation Procedure for Conformance Testing for CEFSM Specified Protocols. In *Proceedings of the IFIP TC6 11th International Workshop on Testing Communicating Systems*, IWTCS, pages 275–290, Deventer, The Netherlands, 1998. Kluwer, B.V.

[19] C. Bourhfir, R. Dssouli, M. Aboulhamid, and N. Rico. A test case generation tool for conformance testing of SDL systems. In *SDL Forum*, pages 405–420, 1999.

[20] J. Bozic and F. Wotawa. Model-based testing - from safety to security. In *9th Workshop on Systems Testing and Validation (STV'12). Proceedings*, pages 9–16, October 2012.

[21] D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. *J. ACM*, 30(2):323–342, April 1983.

[22] L. Briand, J. Cui, and Y. Labiche. Towards automated support for deriving test data from uml statecharts. In *The Unified Modeling Language. Modeling Languages and Applications*, pages 249–264. Springer, 2003.

[23] H. Briggs. Cryosat rocket fault laid bare. *http://news.bbc.co.uk/go/pr/fr/-/2/hi/science/nature/4381840.stm*, 2005.

[24] G. Caire, M. Cossentino, A. Negri, A. Poggi, and P. Turci. Multi-agent systems implementation and testing. In *In Fourth International Symposium: From Agent Theory to Agent Implementation*, pages 14–16, 2004.

[25] J. Cheesman and J. Daniels. *UML Components A Simple Process for Specifying Component Based Software.* Addison-Wesley, 2001.

[26] S. Dalal, A. Jain, N. Karunanithi, J. Leaton, C. Lott, G. Patton, and B. Horowitz. Model-Based Testing in Practice. In *ICSE*, pages 285–294, 1999.

[27] J. de Muijnck-Hughes and I. Duncan. Issues affecting security design pattern engineering. *Proceedings of Cyberpatterns*, pages 54–61, 2013.

[28] K. Derderian, R. Hierons, M. Harman, and Q. Guo. Estimating the feasibility of transition paths in extended finite state machines. *Automated Software Engineering*, 17(1):33–56, 2010.

[29] G. Di Guglielmo, F. Fummi, C. Marconcini, and G. Pravadelli. A pseudo-deterministic functional atpg based on efsm traversing. In *null*, pages 70–75. IEEE, 2005.

[30] T. Dinh-Trong, N. Kawane, S. Ghosh, R. France, and A. Andrews. A tool-supported approach to testing uml design models. In *Engineering of Complex*

Computer Systems, 2005. ICECCS 2005. Proceedings. 10th IEEE International Conference on, pages 519–528. IEEE, 2005.

[31] M. Eby, J. Werner, G. Karsai, and A. Ledeczi. Integrating security modeling into embedded system design. *Proceedings of the 14th Annual IEEE Intl. Conf. And W'shops on the Engineering of Computer-Based Systems*, pages 221–228, 2007.

[32] A. Ek, J. Grabowski, D. Hogrefe, R. Jerome, B. Koch, and M. Schmitt II. Towards the industrial use of validation techniques and automatic test generation methods for SDL specifications. In *SDL Forum*, pages 245–260, 1997.

[33] K. Epstein and B. Elgin. Network security breaches plague NASA. pages 1–11, November 2008.

[34] M. Felderer, B. Agreiter, P. Zech, and R. Breu. A classification for model-based security testing. *IARIA, 2011*, pages 109–114, 2011.

[35] D. Firesmith. A taxonomy of security-related requirements. *Software Engineering Institute*, pages 1–11, 2005.

[36] G. Friedman, A. Hartman, K. Nagin, and T. Shiran. Projected state machine coverage for software testing. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 134–143. ACM, 2002.

[37] A. Gario, A. Andrews, and S. Hagerman. Testing of safety-critical systems: An aerospace launch application. In *2014 IEEE Aerospace Conference*, pages 1–17, March 2014.

[38] M. Gegick and L. Williams. Matching attack patterns to security vulnerabilities in software-intensive system designs. In *Proceedings of the 2005 workshop on*

Software engineering for secure systems building trustworthy applications, SESS '05, pages 1–7, New York, NY, USA, 2005. ACM.

[39] S. Ghosh, R. France, C. Braganza, N. Kawane, A. Andrews, and O. Pilskalns. Test adequacy assessment for uml design model testing. In *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*, pages 332–343. IEEE, 2003.

[40] M. Gogolla, J. Bohling, and M. Richters. Validation of uml and ocl models by automatic snapshot generation. *In UML*, pages 265–279, November 2003.

[41] K. He, Z. Feng, and X. Li. An attack scenario based approach for software security testing at design stage. In *ISCSCT (1)*, pages 782–787, 2008.

[42] R. He and M. Lacoste. Applying component-based design to self-protection of ubiquitous systems. *ICPS Proceedings of the 3rd ACM Workshop on Software Engineering for Pervasive Services*, pages 9–14, 2008.

[43] O. Henniger, M. Lu, and H. Ural. Automatic Generation of Test Purposes for Testing Distributed Systems. In *Formal Approaches to Software Testing*, volume 2931, pages 1105–1105. Springer Berlin/Heidelberg, 2004.

[44] A. Hessel and P. Pettersson. A Global Algorithm for Model-Based Test Suite Generation. *Electronic Notes in Theoretical Computer Science*, 190(2):47–59, 2007.

[45] A. Iqbal, M.and Arcuri and L. Briand. Empirical investigation of search algorithms for environment model-based testing of real-time embedded software. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 199–209, New York, NY, USA, 2012. ACM.

[46] J. Jrjens. Model-based security testing using umlsec: A case study. *Electr. Notes Theor. Comput. Sci.*, 220(1):93–104, 2008.

[47] J. Jurjens. Umlsec: Extending uml for secure systems development. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, UML '02, pages 412–425, London, UK, UK, 2002. Springer-Verlag.

[48] A. Kalaji, R. Hierons, and S. Swift. A search-based approach for automatic test generation from extended finite state machine (efsm). In *Testing: Academic and Industrial Conference-Practice and Research Techniques, 2009. TAIC PART'09.*, pages 131–132. IEEE, 2009.

[49] K. Kang and S. Son. Towards security and qos optimization in real-time embedded systems. *ACM SIGBED Review, Special Interest Group on Embedded Systems, SIGBED Review, Vol. 3, No. 1 (2006))*, 2006.

[50] K. Karppinen, R. Savola, M. Rapeli, and E. Tikkala. Security objectives within a security testing case study. *ARES Proceedings of the The Second International Conference on Availability, Reliability and Security*, pages 1060–1065, 2007.

[51] L. Khelladi, Y. Challal, A. Bouabdallah, and N. Badache. On security issues in embedded systems: Challenges and solutions. *International Journal of Information and Computer Security 2, 2*, pages 140–174, 2008.

[52] Y. Kissoum and Z. Sahnoun. Test cases generation for multi-agent systems using formal specification.

[53] P. Kocher, R. Lee, G. McGraw, A. Raghunathan, and S. Ravi. Security as a new dimension in embedded system design. *IEEE Design Automation Conference*, 2004.

[54] G. Kovács, Z. Pap, and G. Csopaki. Automatic Test Selection Based on CEFSM Specifications. *Acta Cybern.*, 15(4):583–599, Dec. 2002.

[55] C. Larman. *Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development*. Pearson Education India, 2005.

[56] D. Lee and M. Yannakakis. Principles and Methods of Testing Finite State Machines- A Survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[57] R. Lefticaru and F. Ipate. Automatic state-based test generation using genetic algorithms. In *synasc*, pages 188–195. IEEE, 2007.

[58] B. Lerner, S. Christov, L. Osterweil, R. Bendraou, U. Kannengiesser, and A. Wise. Exception handling patterns for process modeling. *IEEE Transactions on Software Engineering*, 36(2):162–183, March 2010.

[59] J. Li and W. Wong. Automatic test generation from communicating extended finite state machine (cefsm) based models. *ISORC '02 Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2002.

[60] G. Luo, R. Dssouli, G. von Bochmann, P. Venkataram, and A. Ghedamsi. Generating synchronizable test sequences based on finite state machine with distributed ports. In *Protocol Test Systems*, pages 139–153, 1993.

[61] A. Marback, H. Do, K. He, S. Kondamarri, and D. Xu. A threat model-based approach to security testing. *Software - Practice and Experience*, 43:241–258, February 2012.

[62] T. Massey, P. Brisk, F. Dabiri, and M. Sarrafzadeh. Delay aware, reconfigurable security for embedded systems. *Proceedings of the ICST 2nd international conference on Body area networks*, 2007.

[63] G. McGraw and B. Potter. Software security testing. *IEEE Security and Privacy archive Volume 2 , Issue 5 September 2004*, pages 81–85, 2004.

[64] T. Mouelhi, F. Fleurey, B. Baudry, and Y. Le Traon. *A Model-Based Framework for Security Policy Specification, Deployment and Testing*, volume 5301 of *Lecture Notes in Computer Science*, chapter 38, pages 537–552. Springer Berlin / Heidelberg, 2009.

[65] J. Offutt and A. Abdurazik. Generating tests from uml specifications. In *The Unified Modeling Language*, pages 416–429. Springer, 1999.

[66] K. Patel and S. Parameswaran. Shield: A software hardware design methodology for security and reliability of mpsocs. *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE Volume , Issue, 8-13*, pages 858–861, 2008.

[67] T. Pender. *UML Bible*. John Wiley and Sons Inc, New York, NY, USA, 2003.

[68] K. Peralta, A. Orozco, A. Zorzo, and F. Oliveira. Specifying security aspects in uml models. In *ACM/IEEE 11th International Conference on Model Driven Engineering Languages and System, Toulouse, França, Proceedings of the Workshop on Modeling Security (MODSEC08 1: 1-10*, 2008.

[69] S. Peterson and P. Faramarzi. Exclusive: Iran hijacked us drone, says iranian engineer. *csmonitor.com*, 2011.

[70] O. Pilskalns, A. Andrews, S. Ghosh, and R. France. Rigorous testing by merging structural and behavioral uml representations. In *The Unified Modeling Language. Modeling Languages and Applications*, pages 234–248. Springer, 2003.

[71] O. Pilskalns, A. Andrews, A. Knight, S. Ghosh, and R. France. Testing uml designs. *Information and Software Technology*, 49(8):892–912, 2007.

[72] O. Pilskalns, G. Uyan, and A. Andrews. Regression testing uml designs. In *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*, pages 254–264. IEEE, 2006.

[73] S. Ravi, A. Raghunathan, and S. Chakradhar. Tamper resistance mechanisms for secure, embedded systems. *Proceedings of the 17th International Conference on VLSI Design (2004)*, page 604, 2004.

[74] M. Rehman, F. Jabeen, A. Bertolino, and A. Polini. Software component integration testing: A survey. *A Survey, Journal of Software Testing, Verification and Reliability*, 2005.

[75] J. Rumbaugh, I. Jacobson, and G Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Boston, MA, USA, 2005.

[76] K. Sabnani and A. Dahbura. A protocol test generation procedure. *Computer Networks and ISDN systems*, 15(4):285–297, 1988.

[77] P. Salas, P. Krishnan, and K. Ross. Model-based security vulnerability testing. *Software Engineering Conference, 2007. ASWEC 2007. 18th Australian*, pages 284–296, 2007.

[78] I. Schieferdecker. Model-based testing. *IEEE Software*, 29(1):14–18, 2012.

[79] I. Schieferdecker, J. Grossmann, and M. Schneider. Model-based security testing. In Alexander K. Petrenko and Holger Schlingloff, editors, *MBT*, volume 80 of *EPTCS*, pages 1–12, 2012.

[80] Z. Shao, C. Xue, Q. Zhuge, M. Qiu, B. Xiao, and E. Sha. Security protection and checking for embedded system integration against buffer overflow attacks via hardware/software. *IEEE Transactions on Computers archive Volume 55 , Issue 4*, pages 443–453, 2006.

[81] G. Shu and D. Lee. Testing security properties of protocol implementations-a machine learning based approach. In *Distributed Computing Systems, 2007. ICDCS'07. 27th International Conference on*, pages 25–25. IEEE, 2007.

[82] B. Smith and L. Williams. On the effective use of security test patterns. In *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*, pages 108–117. IEEE, 2012.

[83] M. Stytz and S. Banks. Dynamic software security testing. *IEEE Security and Privacy archive Volume 4 , Issue 3 May 2006*, pages 77–79, 2006.

[84] L. Tahat, B. Vaysburg, B. Korel, and A. Bader. Requirement-based automated black-box test generation. In *Computer Software and Applications Conference, 2001. COMPSAC 2001. 25th Annual International*, pages 489–495. IEEE, 2001.

[85] H. Thompson. Why security testing is hard. *IEEE Security and Privacy Volume 1 , Issue 4 July 2003*, pages 83–86, 2003.

[86] H. Thompson, J. Whittaker, and F. Mottay. Software security vulnerability testing in hostile environments. In *Proceedings of the 2002 ACM symposium on Applied computing*, SAC '02, pages 260–264, New York, NY, USA, 2002. ACM.

[87] G. Tian-yang, S. Yin-sheng, and F. You-yuan. Research on software security testing. *World Acadmeny of Science, Engineering and Technology*, 69:647–651, 2010.

[88] I. Tondel, M. Jaatun, and J. Jensen. Learning from software security testing. In *Software Testing Verification and Validation Workshop, 2008. ICSTW'08. IEEE International Conference on*, pages 286–294. IEEE, 2008.

[89] A. Tripathy and A. Mitra. Test case generation using activity diagram and sequence diagram. In *Proceedings of International Conference on Advances in Computing*, pages 121–129. Springer, 2013.

[90] B. Tsai, S. Stobart, N. Parrington, and I. Mitchell. An automatic test case generator derived from state-based testing. In *apsec*, page 270. IEEE, 1998.

[91] S. Turpe. Security testing: Turning practice into theory. *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, 2008.

[92] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[93] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.

[94] R. Vaslin, G. Gogniat, J. Diguet, E. Wanderley, R. Tessier, and W. Burleson. A security approach for off-chip memory in embedded microprocessor systems. *Journal of Microprocessors and Microsystems, vol. 33, no. 1 (2009)*, pages 37–45, 2009.

[95] L. Wang, E. Wong, and D. Xu. A threat model driven approach for security testing. In *Proceedings of the Third International Workshop on Software Engineering for Secure Systems*, SESS '07, pages 10–, Washington, DC, USA, 2007. IEEE Computer Society.

[96] J. Werner, M. Eby, J. Mathe, G. Karsai, Y. Xue, and J. Sztipanovits. Integrating security modeling in embedded system design. *Engineering of Computer-Based Systems, 2007. ECBS apos;07. 14th Annual IEEE International Conference and Workshops on the Volume , Issue , 26-29*, pages 221–228, 2007.

[97] J. Whittaker and H. Thompson. Black box debugging. *Queue Volume 1 , Issue 9*, pages 68–74, 2004.

[98] A. Wiesauer and J. Sametinger. A security design pattern taxonomy based on attack patterns. In *International Joint Conference on e-Business and Telecommunications*, pages 387–394, 2009.

[99] G. Wimmel and J. Jürjens. Specification-based test generation for security-critical systems using mutations. In *Proceedings of the 4th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, ICFEM '02, pages 471–482, London, UK, UK, 2002. Springer-Verlag.

[100] F. Xie, G. Yang, and X. Song. Component-based hardware/software co-verification for building trustworthy embedded systems. *Journal of Systems and Software*, Volume 80 , Issue 5:643–654, 2007.

[101] D. Xu. A tool for automated test code generation from high-level petri nets. *Proc. of the 32nd International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets 2011)*, 2011.

[102] A. Yasar, P. Davy, Y. Berbers, and G. Bhatti. Best practices for software security: An overview. *12th IEEE International Multitopic Conference (IEEE INMIC 2008) vol:12*, pages 169–173, 2008.

[103] J. Yoder and J. Barcalow. Architectural patterns for enabling application security. 1997.

[104] S. Zafar and R. Dromey. Integrating safety and security requirements into design of an embedded system. *apsec, 12th Asia-Pacific Software Engineering Conference (APSEC'05)*, pages 629–636, 2005.

Appendix A: Software Threat Types

Software Threat Types				
Threats	Type	How They Work	Means to Deal With Them	Reference Papers
Processing Performance	Physical Limitation	Systems have resource constrains minimizes security	1.Choose basic security primitives. 2.Use of cryptographic algorithms. 3.Meticulous design of software and hardware implementation	[51]
Power Consumption Optimization	Physical Limitation	Energy consumption is influenced bulky encrypted data	1.Use of energy efficient security protocol. 2.Meticulous design of software and hardware implementation 3.Allow behavior adaptation by altering tasks to	[51]

Software Threat Types cont				
Threats	Type	How They Work	Means to Deal With Them	Reference Papers
Viruses, Worms, Trojan Horses, etc.	Software Attacks	Operating Systems may not have boundary checks Malicious code gains access to stack, heap, and function pointers	1.Ensure privacy and integrity at each step of execution. 2.Define safe environment to execute code from 3.Validate software before execution. 4. Remove loopholes that compromise security. 5.Involve hardware support by using a secure co-processor. 6.Verify bootstrapping is secure. 7.Enhance operating system to provide security. 8.Use of cryptographic algorithms. 9.Involve security considerations early in the design. 10. Meticulous software and hardware co-design.	[31] [51] [49] [53] [73] [66] [80] [96] [104]
Timing Constraints	Physical Limitation	Systems have resource constraints	1. Use of a delay-aware heuristic that adapts levels of security to meet timing constraints	[62]

Software Threat Types cont				
Threats	Type	How They Work	Means to Deal With Them	Reference Papers
Eavesdropping	Physical Attack	Attackers use probes to eavesdrop on inter-component communication directly at the level	1.Choose basic security primitives 2.Use of cryptographic algorithms. 3.Meticulous design of software and hardware implementation. 4. Using processors to encrypt/decrypt all data on buses. 5. Employ a hardware architecture that utilizes an off-chip memory security solution.	[51] [94] [73]

Software Threat Types cont				
Threats	Type	How They Work	Means to Deal With Them	Reference Papers
Simple Power Analysis Attacks	Physical Attack	The power profile of cryptographic algorithm computations can be used to gain access to cryptographic key	1.Addition of tamperevident packaging technologies (i.e.seal or enclosure) 2. Employ use of cryptoprocessors with internal tamper circuitry to physically secure data. 3. Meticulous design of software and hardware 4. Using processors to encrypt/decrypt all data on buses. 5. Employ a hardware architecture that uses an off-chip memory security solution	[51] [94] [73]

Software Threat Types cont				
Threats	Type	How They Work	Means to Deal With Them	Reference Papers
Differential Power Analysis Attacks	Physical Attack	Attack uses statistical aphic analysis of the difference between traces to deduce the cryptographic key	1. Addition of tamper-evident packaging technologies (i.e. seal oe enclosure) 2. Employ use of crypto processors with internal tamper circuitry to physically secure data. 3. Meticulous design of software and hardware 4. Using processors to encrypt/decrypt all data on buses. 5. Employ a hardware architecture that uses an off-chip memory security solution	[51] [94] [73]

Software Threat Types cont				
Threats	Type	How They Work	Means to Deal With Them	Reference Papers
Instruction Execution Time Variation Attacks	Side Channel Attack	Cryptographic key can be broken by analyzing number of cycle and data	1. The use of a random clock signal removes determinism makes it more difficult to replicate the clock signal. 2.Introducing noise into power measurement data 3.Using data masking to conceal sensitive data 4.The use of scheduling algorithms which use slack time to run security tasks	[51] [100] [73]

Software Threat Types cont				
Threats	Type	How They Work	Means to Deal With Them	Reference Papers
Performance Optimization Attacks	Side Channel Attack	Optimization may introduce vurnerability in execution paths	1. The use of a random clock signal removes determinism makes it more difficult to replicate the clock signal 2.Introducing noise into power measurement data 3.Use data masking to conceal sensitive data. 4.The use of scheduling algorithms which use slack time to run security tasks	[51] [73] [100]

Software Threat Types cont				
Threats	Type	How They Work	Means to Deal With Them	Reference Papers
Privacy Attacks	Fault Injection Attacks	Attackers use fault injections to break cyrptographic keys	1. The use of a random clock signal removes determinism makes it more difficult to replicate the clock signal. 2.Introducing noise into power measurement data. 3.Using data masking conceal sensitive data. 4.Employ use of crypto processors with internal circuitry to physically secure data	[73]
Pre-Cursor Attacks	Fault Injection Attacks	Attackers use fault injections as a pre-cursor to software attacks	1. The use of a random clock signal removes determinism makes it more difficult to replicate the clock signal. 2.Introducing noise into power measurement data. 3.Using data masking to conceal sensitive data 4.Employ use of crypto processors with internal circuitry to physically secure data 5.Verify bootstrapping is secure	[73]

Software Threat Types cont				
Threats	Type	How They Work	Means to Deal With Them	Reference Papers
Electro-Magnetic Analysis Attacks	Radiation Emission Attacks	Attackers attempt to measure electro-magnetic radiation to recreate screen contents	1. The electro-magnetic field must be kept isolated	[73]
Integrity Attacks	Fault Injection Attacks	Attackers corrupt code or data in memory	1. The use of a random clock signal removes determinism makes it more difficult to replicate the clock signal. 2.Introducing noise into power measurement data. 3.Using data masking to conceal sensitive data. 4.Employ use of crypto processors with internal circuitry to physically secure data	[73]

Table A.1: Software Threats

Appendix B: Software Threat Models

Software Threat Model				
Identify Threats	Understand Threats	Categorize Threats	Mitigation	Test
Data Substitution	Attacker can substitute a requested data block for another without detection	Tampering, Elevation of Privilege Spoofing	The code block and hash are encrypted by the compiler compared at runtime	Through Testing With Attempts to substitute data blocks
Data Injection/ Modification	Attacker can guess location of runtime stack by looking at the data write backs	Tampering, Information Disclosure, Elevation of Privilege or Spoofing, Repudiation, Denial of Service	Use CRC32 as a unique signature in each block of code encrypted	Through Testing With Attempts to locate the stack

Software Threat Model				
Identify Threats	Understand Threats	Categorize Threats	Mitigation	Test
Code Injection/ Execution Disruptions	Attacker tries to replace or modify a section of encrypted instructions without detection	Spoofing, Tampering Information Disclosure Elevation of Privilege	Run Time Code Integrity Checking	Through Testing With Attempts to replace or modify sections of code
Instruction Replay	Attacker replays a set of encrypted instructions from memory without detection	Tampering, Information Disclosure, Elevation of Privilege or Spoofing	Validate that the data contents are from the mem location they were requested from	Through Testing With Attempts to replace a set of instructions
Control Flow Attacks	Attacker can sniff the data bus to gain access to control flow structure	Tampering, Information Disclosure, Elevation of Privilege or Spoofing, Repudiation, Denial of Service	Control flow must be embed-ded and validated by a built in applica-tion	Through Testing With Attempts to alter the control flow

Table B.1: Threat Model