Architectural and Behavioral Analysis for Cyber Security

Kit Siu, Abha Moitra,
Meng Li, Michael Durling,
Heber Herencia-Zapana,
John Interrante, Baoluo Meng
GE Global Research Center
Niskayuna, NY

Cesare Tinelli, Omar Chowdhury,
Daniel Larraz, Moosa Yahyazadeh,
M. Fareed Arif
University of Iowa
Iowa City, IA

Daniel Prince
GE Aviation Systems
Grand Rapids, MI

***Abstract*— The asymmetric nature and ever-increasing degree of sophistication of cyber threats drive the need for assurance of critical infrastructure and systems. Current approaches that can help counter these cyber threats include the application of tools with the ability to analyze system behavior in its most general form and in the presence of wide classes of threats—at design time. In this paper we describe our tool for incorporating cyber security resiliency analysis and recommendations in the system design process that are automated, scalable, provide rich feedback, specify trade-offs and are easy to use by system architects. The architecture models and design knowledge are captured in formalisms that are expressive and amenable to automated reasoning, analysis, and analysis for cyber security. The two main components developed are Model-Based Architectural Analysis and Cyber-resiliency Verifier. The Model-Based Architectural Analysis identifies threats against the architecture and the potential tradeoffs of their mitigations. Threats are identified from attack patterns from Mitre's Common Attack Pattern Enumeration and Classification (CAPEC) and defenses are suggested based on controls from NIST's Security and Privacy Controls list. After the threats and defenses are identified, an attack-defense tree is generated along with cutsets that describe attack scenarios and associated defenses. The identified threats as well as the attack-defense trees are visualized to provide a concise and informative view of the analysis. There may be multiple ways of addressing a threat and the system architect can decide which defense to implement. After the architecture has been revised to be secure, we then consider cyber-resiliency behavioral properties of the system. For this we abstract threat models in terms of an instrumentor that incorporates the effects of the threats. This allows us to aggregate classes of threats with the same effect so that they can be addressed at the effects level as opposed to treating them individually. The cyber-resiliency properties are then verified by an extension of the proven model checker Kind 2 which identifies the system components responsible for a property violation and provides localized diagnostic information. This enables the system architect to rework the vulnerable portions of the design to discharge required resiliency obligations. In case of resiliency property violations, the tool can recommend placement of runtime monitors sufficient to discharge specific proof obligations. Finally, our tool automatically generates test cases and procedure to check the conformance between the design and implementation.**

Keywords— cyber security, attack-defense trees, behavioral properties, test cases.

I. INTRODUCTION

The asymmetric nature and ever-increasing degree of sophistication of cyber threats drive the need for assurance of critical infrastructure and systems. The conventional belief was that cyberattacks on critical infrastructures designed as embedded systems are of no concern because they were seldom connected to the network from which these attacks were enabled. However, attackers have learned to bridge air gaps – even the most remote and physically guarded infrastructures can be attacked [1]. In short, embedded systems are now subject to cyberattacks, either as the end goal of the cyber assailant or as a means to a greater end. There is a critical need to protect and defend embedded systems in a cyber-context. Current design practice to secure embedded systems include development of software using cyber best practices, adapting mechanisms from information technology (IT) systems, and penetration testing followed by patching. However, these methods may not be completely effective [2].

Better approaches that can help counter cyber threats is the application of tools to analyze system behavior in its most general form and in the presence of wide classes of threats – at design time. Cyber resiliency means a system's ability to tolerate cyberattacks, much in the same way that safety critical systems are measured by their tolerance against random faults. Systems engineers are responsible for producing designs with a required set of functional and non-functional properties. Non-functional properties include those related to safety, performance, availability, maintainability, etc. The system designer is responsible for managing conflicts and making tradeoffs between the desired non-functional properties. Our goal is to develop a tool that allows system engineers to design-in cyber resiliency and manage tradeoffs, as they do the other nonfunctional properties when designing complex embedded computing systems.

Most tools and techniques that offer this type of design-time rigor have been developed in the cyber physical systems / formal methods communities. We propose to close the gap between the common practice of design for cyber resiliency in industry today and the power of tools, formalisms, and knowledge developed in the academic and research communities in the last decades. In this paper we describe our tool for incorporating cyber security resiliency analysis and recommendations in the system design process that are automated, scalable, provide rich feedback, specify trade-offs and are easy to use by system architects. Our project and tool is called Verification Evidence and Resilient Design in Anticipation of Cybersecurity Threats (VERDICT), developed as part of the DARPA Cyber Assured Systems Engineering (CASE) program. The VERDICT team is composed of researchers and engineers from General Electric Research

(GRC), General Electric Aviation Systems (GEAS), and the University of Iowa. The team created an unmanned vehicle-based challenge problem to support the tool development. The example project included AADL (Architecture Analysis & Design Language [3]) models, mission and cyber requirements, formal cyber properties and behavioral models. Currently, the VERDICT tool is able to demonstrate generation of an attack-defense tree with quantifiable likelihood of successful attack metrics, prove behavioral models met formal cyber properties, identify the need for and place run time monitors, and generate cyber test cases. This document includes detailed technical descriptions of the tool functionality and an illustrative example of how the tool works and what artifacts are generated. Section II of the paper describes the Hawkeye UAV model that the team created as an experimental platform. Section III describes the Model-Based Architectural Analysis capability. Section IV introduces the Cyber-resiliency Verifier, Section V describes our approach for Test Generation and Section VI is the Conclusion.

II. THE HAWKEYEUAV MODEL

To demonstrate the effectiveness of the VERDICT tools, we have used a simplified UAV system as a case study. Our UAV system architecture is inspired by the PX4 quadcopter system [4] which was heavily used in the DARPA High-Assurance Cyber Military Systems (HACMS) program. The high-level architecture of our UAV system, which we call the HawkeyeUAV, is shown in Fig. 1.

The HawkeyeUAV system can operate in either manual or autonomous flight mode. In manual flight mode, the responsibility of navigation is deferred to a remote pilot component which communicates with the UAV system through a radio-frequency channel.

For autonomous flight mode, the Mission Planner is pre-programed with mission-related tasks and relays that to the Flight Controller which is mainly responsible for managing the successful completion of the mission-related tasks. The Flight Controller provides the navigator with the waypoints to navigate the UAV. The current location information is provided to the Navigator by the Position Estimator. The Position Estimator receives inputs from sensors and estimates the current location of the UAV.

The Navigator is responsible for guiding the UAV to the next waypoint from the current location. It provides high-level direction to the Mixer which compiles it down to low-level instructions that can be understood by the Actuators so that it can send the proper signals to the actuators. The Navigator notifies the Flight Controller whenever it reaches the designated waypoint. In that case, the Flight Controller communicates the appropriate payload commands (e.g., capture an image) to the Actuators.

The HawkeyeUAV system is equipped with a global positioning system (GPS) sensor. The UAV system is battery-powered and has a Health Check Monitor for the Battery. This component will raise a warning when the battery level goes below a predefined level.

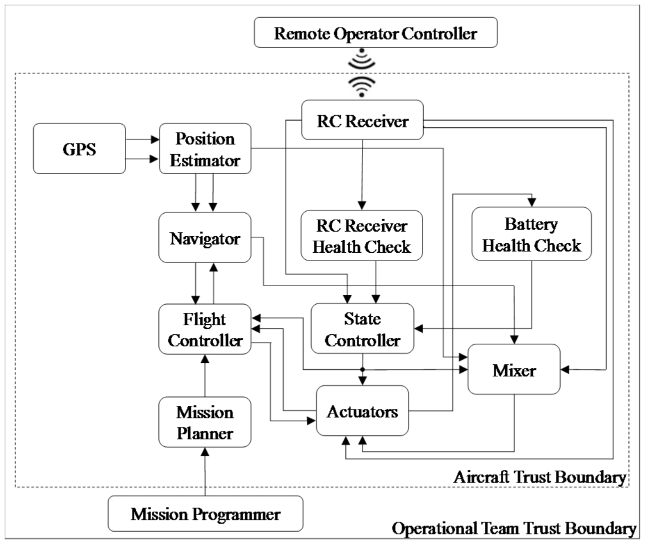

Figure 1. HawkeyeUAV Architecture.

For our case study, we consider two actuators. The Camera actuator is responsible for capturing images. The Motor and Flight Surfaces actuators, on the other hand, are responsible for carrying out the actual movements of the HawkeyeUAV.

In this paper, we will be analyzing the HawkeyeUAV against the mission requirement, "The UAV shall perform reconnaissance of designated civilian habitat locations." This mission requirement is decomposed into mission-level cyber-resiliency requirements, two of which are "1.1 *The UAV shall be resilient to loss of ability to perform reconnaissance of designated civilian habitat*" and "1.2 *The UAV shall be resilient to maliciously commanded improper reconnaissance*."

III. MODEL-BASED ARCHITECTURAL ANALYSIS (MBAA)

The purpose of analyzing the architecture is to determine if the system is resilient enough to perform a given mission. We start at the architecture level because a lot can be analyzed just from knowing what types of components are in the system, how they are connected, what type of information flows between them, and their exposure to threats from outside the system's trusted boundary. Our analysis highlights the area of the architecture where defenses are either missing/inappropriate, or insufficient rigor was applied compared to the level of severity if an attack was successful. To analyze the architecture, we do a top-down approach to see what combination of events lead to an undesired state of a system. We developed a Security Threat Evaluation and Mitigation (STEM) tool that takes in architectural information about a system and identifies the possible threats and their corresponding mitigations. They are then analyzed using attack-defense trees. Based on prior work by Kordy, et al. [5] [6] attack-defense tree is a better representation of a system over attack trees because the latter only captures attack scenarios and does not model the interaction between attacks and the defenses that could be put in place to guard against the attacks. We further extended their work to include guidelines and considerations from DO-356A [7]so that the terminology used in the tree is relevant to the aviation industry.

A. *Identifying Security Threats and Mitigations*

Security Threat Evaluation and Mitigation (STEM) function takes in architectural information about a system as input and identifies the possible threats and their corresponding mitigations. The threats are identified in terms of Mitre's Common Attack Pattern Enumeration and Classification (CAPEC) [8] and the defenses are from NIST's Security and Privacy Controls [9], though any attack and defense databases can be used. NIST 800-53 defenses are described at a high level, but by explicitly defining profiles in terms of NIST controls and using the profile as a "punch list", variances between products and suppliers in implementation and subjectivity in tailoring controls lists are removed..

In the HawkeyeUAV shown in Fig. 1, for the RC Receiver, since it receives Wi-Fi communication from outside the aircraft trust boundary (TB), it is vulnerable to several CAPECs including CAPEC-151 (Identity Spoofing). To mitigate CAPEC-151, we have identified a list of NIST defenses for which we will construct rules detailed in an upcoming section. The next question is how do we implement these NIST defenses? Suitable implementation of encryption mitigates CAPEC-151 vulnerability. Adding in encryption would be reflected as an implementation property. In general, the NIST defenses can be related to architectural changes (e.g. SC-29 Heterogeneity which we will see later) as well as putting in place appropriate controls. This relationship is depicted in Fig. 2.

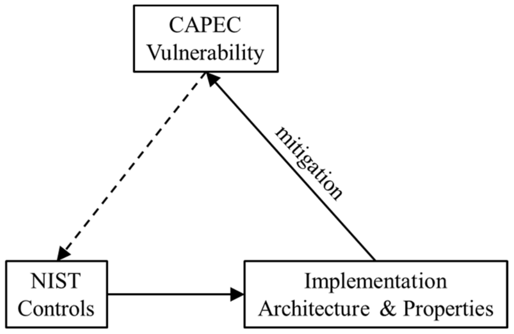

Figure 2. Relationship between CAPEC Vulnerabilities, NIST Controls, and Implementation Properties.

STEM uses a semantic model and rules to identify the vulnerabilities and defenses for an attack scenario. The model and rules in STEM are authored in Semantic Application Design Language (SADL) [10] [11]. It is a controlled-English language that maps directly into the Web Ontology Language (OWL) [12]. It also contains additional constructs to support rules, queries, testing, debugging, and maintaining of knowledge bases. Besides being a language, SADL is also an integrated development environment (IDE) for building, viewing, exercising and maintaining semantic models over their lifecycle. The SADL grammar is implemented in Xtext [13], which also provides the features of a modern IDE including semantic coloring, hyperlinking, outlining, content assistance, etc. The SADL grammar falls into two sections: it supports declaring the ontological concepts of a domain (classes, properties, individuals, property restrictions, and imports), and expressions that may appear in queries, rules, and tests. We assume that the reader is familiar with defining a semantic model, so we will just focus on the STEM rules.

The STEM rules can be classified into 3 categories: 1) rules to identify CAPEC vulnerabilities, 2) rules to identify NIST defenses, and 3) rules that relate NIST defenses to implementation properties of an attack scenario. Based on component properties and Mitre's CAPECs list, we have authored a number of rules to identify applicable CAPECs to a component. Here is one rule that was authored for CAPEC-151 which is titled "Identity Spoofing".

```
Rule Vul-CAPEC-151
if insideTrustedBoundary of a Subsystem is true and
   wifiFromOutsideTB of the Subsystem is true
then there exists (a CIAIssue with ciaIssue Integrity,
                    with affectedComponent the Subsystem,
                    with capec "CAPEC-151",
                    with capecDescription "Identity Spoofing"
                    with likelihoodOfSuccess 1.0) and
  capecString of the Subsystem is "CAPEC-151:Identity Spoofing").
```

For CAPEC-151 we have identified 8 NIST defenses in total. In the rule snippet below, we just show the first two (IA-2, IA-3) of the 8 defenses.

```
Rule Def-CAPEC-151
if ciaIssue of a CIAIssue is Integrity and
  capec of the CIAIssue is "CAPEC-151" and
  affectedComponent of the CIAIssue is a Subsystem
then there exists (a Protection with ciaIssue Integrity,
                    with capecMitigated "CAPEC-151",
                    with capecDescription "Identity Spoofing",
                    with affectedComponent the Subsystem,
                    with defense "IA-2",
                    with protectionDescription "Identification and
                         Authentication (Organizational Users)")
    and there exists (a second Protection with ciaIssue Integrity,
                         with capecMitigated "CAPEC-151",
                         with capecDescription "Identity Spoofing",
                         with affectedComponent the Subsystem,
                         with defense "IA-3",
                         with protectionDescription "Device
                              Identification and Authentication")
          ...
```

Mitigation of CAPEC-151 can be achieved by using suitable encryption. We capture this by the implementation property *allWifiEncFromOutsideTB* with an associated DAL value. The selection of the property name is motivated by the fact that in the system architecture we want to ensure that all Wi-Fi communication for a component with outside the trusted boundary is encrypted.

```
Rule Encryption
if oneOf(defense of a Protection, "IA-2", "IA-3", "IA-7", "SC-8",
                    "SC-10", "SC-12", "SC-13", "SC-23") and
      affectedComponent of the Protection is a Subsystem and
      val of allWifiEncFromOutsideTB of the Subsystem is true
then implProperty of the Protection is "encryption" and
    dal of the Protection is dal of allWifiEncFromOutsideTB
                         of the Subsystem and
    addressed of the Protection is true.
```

Once the vulnerabilities and defenses are identified, they are analyzed using an attack-defense tree.

B. *Analyzing the Sufficiency of Defense*

Attack-defense tree is one way to analyze the sufficiency of mitigations in preventing the success of identified attacks. The top node of the tree represents an attacker's goal. The standard on Airworthiness Security Methods and Considerations DO-356A [7] provides guidance on the acceptable level of risk for each level of severity, summarized in Table I. We quantify an attacker's goal by mapping it to a level of severity should that goal be achieved. We put in significant effort to define precisely the qualitative and quantitative aspects of the rest of the tree, summarized in Table II. Attack-defense trees are made up of two types of nodes – attack nodes and defense nodes. We always assign attack nodes a value of 1 (the highest) for likelihood of success of attack because according to [14] the issue with deciding the likelihood of various attacks is that "the risk values may be different for different researchers according to the information available and level of analysis. Hence, more emphasis should be put on countermeasures for threats which receive high priority."

TABLE I. CONVERSION FROM LEVELS OF SEVERITY TO ACCEPTABLE RISK LEVEL

Levels of Severity	***Acceptable Risk Level***
Catastrophic	1e-9
Hazardous	1e-7
Major	1e-5
Minor	1e-3
No Safety Effect	1

Each defense consists of a technical security mitigation implemented in the design or a physical security measure that reduces the exposure of the system being designed. Each defense is scored according to a predefined scale for the amount of credit that can be taken for the design rigor put into a technical mitigation or the level of exposure reduction provided by a physical security measure. These predefined scales are tuned to encourage an appropriate level of design rigor for the severity of events that could occur, or, in the case of physical security measures, an appropriate level or protection provided by the security measure. Appendix F of DO-356A [7] proposes a "Level of Protection" score based on the development rigor applied to the security measure. Development rigor can be thought of as a set of process activities that "provide confidence in the elimination of errors having a potential security issue that might be introduced during the development process". Since our tree is an attack focused tree, the score for level of rigor for each defense is converted to a likelihood of failure for that defense. Therefore, a defense would be considered successful against an attack if an appropriate set of controls was applied with sufficient level of rigor.

In our attack-defense tree we distinguish between AND/OR gates for attacks and d-AND/d-OR gates for defenses. The reason for this distinction is to preserve flexibility for choosing the quantitative measure for attacks and defenses. Another reason is to preserve the structure of the defense profile in the attack-defense tree; when simplifying the attack-defense tree the AND/OR gates for attacks do not get collapsed with the d-AND/d-OR gates for defenses. Further intuition regarding defense gates and defense profile is that if a profile requires both defense1 and defense2, then the resulting measure is the weaker of the two. If a profile specifies either defense1 or defense2, then the measure is to encourage a designer to focus on the more rigorous defense. Another gate we have in our tree is a C gate to combine attacks and defenses. The output of this gate is measured in terms of likelihood of success of an attack. Therefore, the likelihood of success of an attack is conditional on the defense likelihood of failure.

Following the definitions in Table II, an attack-defense tree can be formalized and computed quantitatively following the measures defined. In the next section, we describe the modeling framework.

TABLE II. ATTACK-DEFENSE TREE QUANTITATIVE MEASURES

Nodes / Gates	**Measure**		**Example**
Attack, $a1$	Measured in terms of likelihood of success		$M(a_1) = 1$
Defense, $d1$	Measured in terms of design assurance level (DAL)		$M(d_1) = DAL\ E = 1$ $M(d_2) = DAL\ B = 7$ $M(d_3) = DAL\ A = 9$
	Assurance	***Score***	
	DAL E	1	
	DAL D	3	
	DAL C	5	
	DAL B	7	
	DAL A	9	
AND [*a1*, *a2*]	min (*likelihood1*, *likelihood2*)		min(1,1) = 1
OR [*a1*, *a2*]	max (*likelihood1*, *likelihood2*)		max(1,1) = 1
dAND [*d1*, *d2*]	min (DAL A, DAL B)		min(9,7) = 7
dOR [*d1*, *d2*]	max (DAL A, DAL B)		max(9,7) = 9
NOT (*d1*)	$1e^{-M(d)}$		$\sim M(d_2) = 1e^{-7}$ = 0.0000001
Combine attacks and defenses, *C* (*a*, *d*) = *AND*[*a*, *NOT*(*d*)]	min (*likelihood*, $1e^{-M(d)}$)		$C(a_1, d_2)$ $= \min(1, 1e^{-7})$ $= 1e^{-7}$

C. Tree Representation

We use a compositional, model-based framework to generate the attack-defense trees. The original framework [15] was developed for fault analysis; we have extended the framework to analyze security. The key contribution of our model-based framework is that we do not require the end-user to construct trees directly. Instead, the trees are automatically generated based on attack propagation information defined at the component level.

The example below shows a GPS library component. This component has two outputs named GPS_dir and GPS_pos. Lines 4-10 contain information related to faults, which we will not cover in this paper. Lines 11-17 contain information related to attacks. The attacks modeled are loss of availability (loa) and loss of integrity (loi). The attack events are the vulnerabilities identified by STEM. For example, line 14 is an attack formula that says "CAPEC-601" affects the "loa" of the component output "GPS_dir". Lines 18-21 contain information on the component's defenses. The defense events are identified by STEM. STEM also reports the defense design assurance levels (DAL), which are listed in the defense rigors. The defense profiles describe how the defenses are applied to which vulnerability. For example, line 20 says that to counter CAPEC-148 (content spoofing), "heterogeneity" is applied to the system, meaning the GPS is used in conjunction with other sensors. To counter CAPEC-601 (jamming), line 21 says to add "wirelessLinkProtection".

```
 1 { name              = "GPS";
 2   input_flows       = [];
 3   output_flows      = ["GPS_dir"; "GPS_pos"; ];
 4   faults            = ["loa"; "ued"];
 5   basic_events      = ["LoA"; "Ued"];
 6   event_info        = [(1e-7, 1.0); (1e-7, 1.0)];
 7   fault_formulas    = [(["GPS_dir"; "loa"], F["LoA"]);
 8                        (["GPS_pos"; "loa"], F["LoA"]);
 9                        (["GPS_dir"; "ued"], F["Ued"]);
10                        (["GPS_pos"; "ued"], F["Ued"])];
11   attacks           = ["loa"; "loi"; ];
12   attack_events     = ["CAPEC-148"; "CAPEC-601"; ];
13   attack_info       = [1.0; 1.0; ];
14   attack_formulas   = [(["GPS_dir"; "loa"], A["CAPEC-601"]);
15                        (["GPS_dir"; "loi"], A["CAPEC-148"]);
16                        (["GPS_pos"; "loa"], A["CAPEC-601"]);
17                        (["GPS_pos"; "loi"], A["CAPEC-148"]); ];
18   defense_events    = ["heterogeneity"; "wirelessLinkProtection";];
19   defense_rigors    = [7; 7;];
20   defense_profiles  = [("CAPEC-148", D["heterogeneity"];);
21                         ("CAPEC-601", D["wirelessLinkProtection"]) ];
22 };
```

All the components of the HawkeyeUAV from Fig. 1 were put in our modeling framework. To analyze the mission-level cyber-resiliency requirement, "*The UAV shall be resilient to maliciously commanded improper reconnaissance*," we identified the components and outputs that impact it and the type of impact in terms of confidentiality, integrity, and availability (CIA). Table III summarizes the components and outputs along with the CIA effects for this requirement.

TABLE III. COMPONENTS, OUTPUTS, AND CIA EFFECTS FOR OUR MISISON-LEVEL CYBER-RESILIENCY REQUIREMENTS

Requirements	***Component***	***Output***	***CIA***
1.1 *The UAV shall be resilient to loss of ability to perform reconnaissance of designated civilian habitat.*	Flight Controller	payloadCmd	Availability, Integrity
	Position Estimator	currentPos	Availability, Integrity
	Position Estimator	currentDir	Availability, Integrity
1.2 *The UAV shall be resilient to maliciously commanded improper reconnaissance.*	Flight Controller	payloadCmd	Integrity
	Position Estimator	currentPos	Integrity
	Position Estimator	currentDir	Integrity

In addition to identifying the components of the system that impacts this requirement, the end-user also determines the level of severity (based on categories from Table II). The severity of the first requirement is Hazardous with an acceptable level of risk = 1e-7, and the second is Major with an acceptable level of risk = 1e-5.

Once the architecture is modeled using our framework, the analyses are performed automatically. Several artifacts are generated. The first is a tree showing the current system without any defenses/mitigations applied. This is shown in Fig. 3a for Requirement 1.1. We also calculate the likelihood of attack, which is 1.0 in this case when no defenses are implemented. Another artifact we generate is an attack-defense tree if all the recommended defenses were applied to the system. Recall that

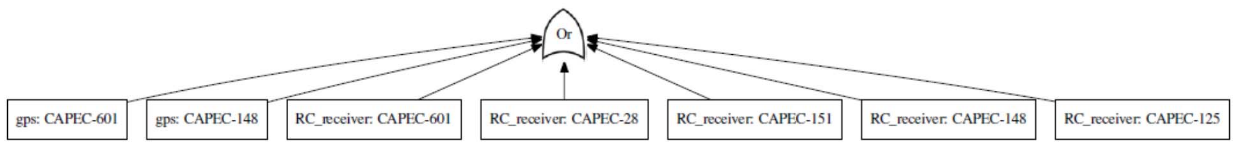

Figure 3a. Analysis of Requirement 1.1 without any defenses applied.

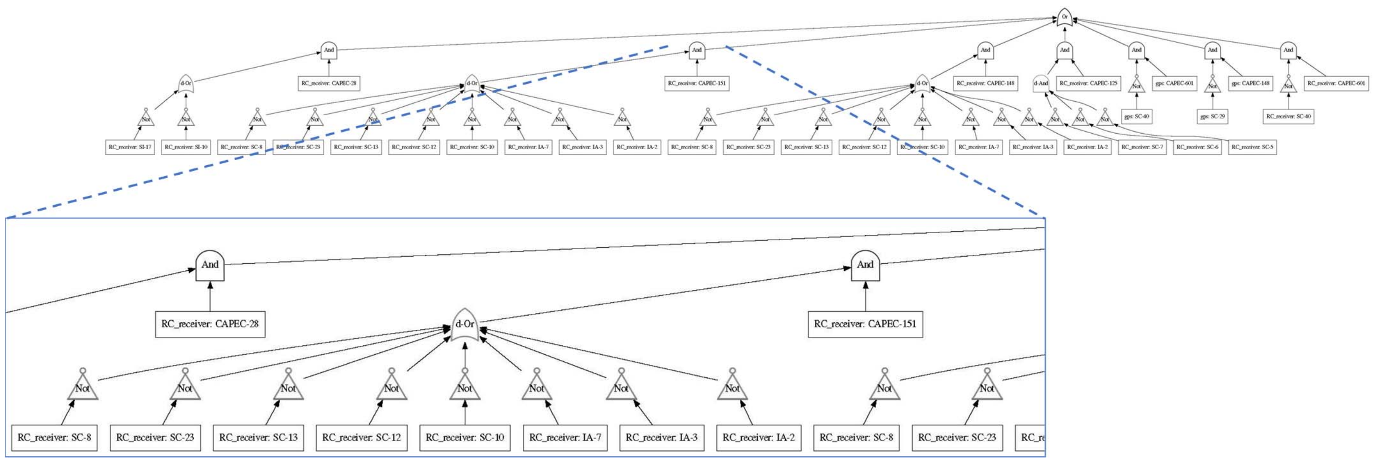

Figure 3b. Analysis of Requirement 1.1 -what the analysis would look like if the recommended defenses were applied (with a zoomed-in portion showing CAPEC-151).

the defenses are those identified by STEM as mitigations against the attacks. The tree with the recommended defenses applied is shown in Fig. 3b, with a zoomed-in portion showing CAPEC-151. The calculated likelihood of success of an attack for this tree is 1e-7, which is within the acceptable level for this requirement.

IV. Cyber-Resiliency Verifier (CRV)

VERDICT's CRV component complements the analysis capability of MBAA by enabling it to reason also about cyber-resilience properties of a system design. To achieve this, CRV considers the system architecture together with an abstract design model of the system. The design model, in our context, enhances the architectural model by including a formal specification of each system component at a level of abstraction that is suitable for cyber-resiliency analysis. The component-level behavior is abstract in the sense that it does not necessarily capture all the fine-grained behavioral details of the component in question. For instance, for the analysis performed by CRV of an unmanned aerial vehicle (UAV) system, it may be sufficient to just model that the UAV moves from one waypoint to another without capturing details regarding how it precisely navigates. Abstractly capturing the component-level behavior also allows CRV to take advantage of automated formal reasoning techniques such as model checking while avoiding the scalability challenges often incurred by such formal techniques (i.e., state-space explosion). Roughly speaking, CRV's analysis can be viewed as performing model checking of the system design model with respect to cyber-resiliency properties while considering an adversarial environment. In the context of a UAV system, for instance, the environment can be viewed as the entity responsible for providing the correct values for the necessary inputs (e.g., speed, location, and altitude of the UAV) required for carrying out the relevant mission goal. An adversarial environment, on the contrary, may feed the UAV wrong values for these input parameters (e.g., wrong location information) modeling an attack which may severely impair UAV's capability to carry out its mission.

Considering the design model enables CRV to more precisely identify or rule out security vulnerabilities of a system beyond what was just analyzed at the architecture level. For example, suppose a system uses a GPS component for navigation. By only inspecting the architecture of a system, the best we could do at the architectural level is to identify that the system is possibly vulnerable to a GPS spoofing attack. Even if the system uses a robust GPS sensor to resist spoofing attacks, due to the lack of behavioral details in the architecture we would not be able to rule out such attacks. Thanks to its access to the design model, however, CRV is able to rule out such false positives. Furthermore, it may identify vulnerabilities missed if inspecting only the system architecture. For instance, suppose a system includes other location sensors (e.g., a LIDAR) together with the GPS sensor and uses a majority voting scheme to rule out spoofing attacks under the assumption that an attacker cannot simultaneously compromise/attack all the positioning sensors on-board. Such a system will be deemed resistant against spoofing attacks by architecture analysis tools. However, by the way this security-enhancing mechanism is designed, it can have logical errors resulting in spoofing attacks. For instance, the majority voting scheme design in the above scenario could have a vulnerability that, under certain conditions, makes it adjudicate the spoofed location value to be the correct value. CRV can identify such a situation through a formal analysis of the components' behavior.

Another way that CRV adds to the analysis is in the threat model it uses for its analysis. More precisely, architectural analysis tools only consider previously documented attacks as part of their threat model. In contrast, CRV groups possible attacks based *on their effects* on the system and collectively reasons about them considering an *abstract threat effect* model. For example, buffer/heap overflow, SQL injection, and return-oriented-programming (ROP) attacks, in the worst case, all provide an attacker with the ability to run malicious code in a system without having prior proper privileges. CRV will thus consider "*running malicious code*" as a possible effect instead of considering all known attacks that achieve this effect. Such an approach has a number of clear benefits. First, the number of effects one must consider for covering a wide variety of attacks is substantially smaller than the number of concrete attacks. Second, this approach can capture also unknown or even future attacks if these attacks have an effect already captured in the threat effect model.

CRV is meant to be used in the system design phase. Its workflow is depicted in Fig. 4. The system architect/designer is responsible for providing the system design model as input to CRV. The designer is also responsible for choosing an appropriate threat effect model from a library of such models under which to carry out the necessary formal resiliency analysis. The CRV workflow intentionally separates the system's design model (①) from the adversarial threat effect models (②). This decoupling enables the system designer to develop the design model without having to include aspects of the system's security properties.

Given the system design model, the chosen threat effect model, as well as a number of desired cyber-resiliency properties for the system, CRV first automatically composes the two model to obtain a threat-instrumented model (④.a). Then, its reasoning engine (a model checker) (refer to ④.b) checks the satisfaction of the properties (③) on the threat-instrumented model. Cyber-resiliency properties are provided by the designer

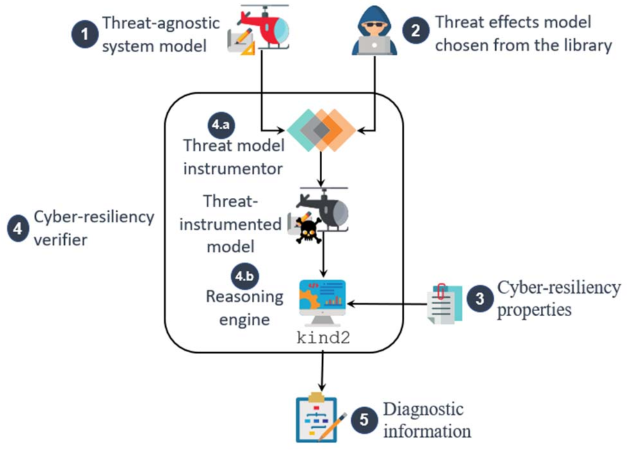
1 Threat-agnostic system model
2 Threat effects model chosen from the library
4.a Threat model instrumentor
Threat-instrumented model
4 Cyber-resiliency verifier
4.b Reasoning engine
kind2
3 Cyber-resiliency properties
5 Diagnostic information

Figure 4. CRV Design.

an expressed in a suitable formal logic. The analysis artifacts (refer to) produces as output by CRV include resilience assurances (proofs), localized diagnosis information to trace possibly vulnerable components in the system and recommend placement of run-time monitors to mitigate threat effects that cause a particular cyber-resiliency property to be violated.

A. Threat Model Instrumentor

The threat model instrumentor of CRV takes as input a system's design model and a threat effect model. It then enhances the system model to consider adversarial actions corresponding to the threats captured by the threat-effect model. The instrumentor's output is a threat-instrumented design model which is amenable to formal reasoning by a model checker.

The class of cyber-resiliency properties currently considered by CRV consists of system integrity properties that can be formalized as temporal safety properties. Considering integrity properties for a system under design is relevant as their violation can cause the system to take unintended actions which may severely jeopardize system safety or security. For instance, a UAV can be led to crash by a successful attack that fools the altitude sensor to provide readings that are much larger than actual ones.

For our threat instrumentation, we have discovered that the effects of threats that violate integrity properties boil down to just a handful of general effects including malicious code injection, software crash, and injection of malicious inputs to a component. Such effects can be modeled by considering the Dolev-Yao adversary model [16], widely used for reasoning about cryptographic protocols. In this adversary model, an attacker can sniff information sent across an insecure channel, and also can modify or inject messages on the channel. All these operations, however, are carried out with natural cryptographic assumptions in mind. For instance, the adversary can read an encrypted message only if the adversary is in possession of the necessary decryption key. We have determined that it is possible to use a Dolev-Yao-style adversary to model all effects of threats that violate integrity properties. For instrumentation, we consider each communication channel connecting two system components of the system to be adversary-controlled. The adversary, at each execution step of the system, can non-deterministically decide to carry out one of the actions available to a Dolev-Yao adversary (i.e., inject, modify, or sniff packets). We also include a null action as one of the possible actions to model the case in which the adversary lets the message go through the channel untampered. This nondeterministic behavioral model of the adversary allows CRV to reason about all possible attack strategies available to the adversary that can result in the violation of the desired cyber-resiliency properties. For usability purposes, we include a library of threat models (e.g., insider threats, remote attacker, corrupted software) to CRV which during instrumentation gets compiled down to its effects.

B. Vulnerability Localization

Once the instrumentation is performed, the resulting enhanced model capturing the adversary behavior along with the desired properties are analyzed with the CRV's backend reasoning engine, the Kind 2 model checker [17]. If the properties are all satisfied, CRV declares the system to be resilient. If instead an attack is found, it is returned to the user by CRV. The attack is presented as an execution of the system, including the attacker actions, which leads to the violation of a cyber-resiliency property. In case of a violation, the designer has the ability to invoke the vulnerability localization feature of CRV, which we discuss next, to identify the vulnerable component(s).

C. Blame assignment

A system is safe with respect to a set of cyber-resiliency properties if its threat-instrumented version satisfies all of them. CRV detects a property violation only if it can construct an execution trace that leads the system to a state where the property in question does not. If the original system behavior, achieved by disabling all attacks, satisfies the property, the violation can only occur in the presence of an attack. In that case, the execution trace represents a concrete attack scenario.

CRV implements a *blame assignment* technique to generate a component-level explanation of the attack which details which system components allow a violation to happen and hence are prone to attack. The blame assignment analysis tries to identify a minimal number of responsible components to assist the system designer in pinpointing the vulnerable parts of the system. Once a vulnerable component is identified, the designer can consider multiple possibilities to address the root cause of the attack. First, one can consider architecture-level changes such as introducing heterogeneity or redundancy (e.g., multiple location sensors whose readings are calibrated to attain a precise location). Second, the designer can change the behavior of the vulnerable component by introducing resiliency measures such as cryptographic authentication and encryption of inputs and outputs. Finally, the designer can include a runtime monitor which observes the system's execution at runtime and tries to identify an attack. When an attack is identified by the runtime monitor, a correction module can be notified which can potentially carry out corrective actions, for instance, scaling malicious inputs to be in the range of valid inputs. CRV has built-in support, discussed next, for automatically suggesting the placement of a runtime monitor. If the designer decides to adopt the runtime-monitoring-based-fix for the design model, he will be required to enhance the behavior of the runtime monitor with a correction module that suggests corrective actions when the monitor detects a transition to an invalid system state.

D. Run-time monitor placement

Once a set of communication channels have been identified as the vulnerable points for an attack, CRV recommends the placement of runtime monitors in these points for the detection of possible violations of the cyber-resiliency properties. The runtime monitors are automatically equipped to identify the violation. Notice that the runtime monitor only detects the transition to an invalid state of the system. It is then the responsibility of the system designer to determine what actual changes are needed to restore the system to a valid state after a violation is detected.

E. *Experimental Evaluation*

We evaluated the capabilities of CRV on the HawkeyeUAV system model shown in Fig. 1. We extended the model with a behavioral specification that considers an autonomous flight mode, a remote pilot mode and a hybrid mode where the pilot and the autonomous flight component work collaboratively towards a successful mission. We expressed the behavior of the different system components in the Lustre dataflow language which is also the input language of the Kind 2 model checker. Since the behavior of some of the components (e.g., the GPS sensor) is dependent on environmental input, for those components we also developed a behavioral model of the environment, a simulator in effect. Let us take the GPS sensor as an example to demonstrate the need of an environmental model. Leaving the GPS sensor's output to be arbitrary would lead the CRV analysis to consider executions where the UAV location values change unrealistically from one execution step to the other (for instance going from a location in North America to one in Asia in the next step). Our GPS simulator takes the current value and the movement into consideration to compute the next location. Additionally, we developed models for the UAV actuators that simulate the effect of different navigation and payload commands (e.g., capture an image).

To capture the behavior of the Battery component, we considered a simulator in which the initial battery level is 100 (for a full charge) and then decrease at each step by a random amount non-deterministically chosen from some range [low, high]. We have another observer component not shown in Fig. 1 which keeps track of the actual location of the UAV based on the current true value and the performed movement. The need of this observer component, keeping track of the UAV's true location, is particularly relevant to verify properties in presence of adversary influence.

For our case study, we considered a simplistic navigation model for the UAV where the UAV moves at constant speed in a two-dimensional space. Locations are expressed as Cartesian coordinates with integer values. We assume that the UAV moves one unit of distance in one unit of time. We also consider the UAV to have the following movements: Go forward one unit of distance in the current direction; Turn right in a predefined angle (in degrees); Turn left in a predefined angle (in degrees). The UAV moves forward only when it is parallel to one of the two axes, with the y-axis placed in the North-South direction and the x-axis in the East-West direction. While this is a very crude approximation of the real movement, it simplifies considerably both the behavioral specification and the work of the model checker. We found it adequate as a proof of concept aimed at showcasing the cyber-resiliency analysis capabilities of the CRV

In our model, a mission is a sequence of tasks. Each task has the form <x, y, A> where x and y are the Cartesian coordinate of the waypoint in which the UAV is required to perform the action designated by A. In our current setting, the action-component can take one of the values from the following domain:

{CAPTURE_IMAGE, NONE}.

The model is parametric in the domain of the action component and can be easily enhanced with additional actions as needed by the system. We also assume each mission has an implicit final task which requires the UAV to return to base after completing all the explicit tasks in the mission. For our case study, without loss of generality, we consider the base to be at the Cartesian coordinate (0,0) and the UAV to be initially at the base and oriented toward the North direction. While in reality our model is parametric also in the location of the base and the initial position and direction of the UAV, we choose the values above here because the simplify the understanding of the different counterexample traces.

We consider the surveillance operation mission in a geographical region. The desired cyber-resiliency properties express the requirement that surveillance is carried out at designated locations and only there. We formally encoded these cyber resilience properties in temporal logic and analyzed them on the HawkeyeUAV system with CRV.

CRV was able to falsify respective some of the resilience properties and provide a counterexample which included a sequence of adversarial actions that lead the system violate those property providing, effectively, a recipe of an attack. CRV generated counterexamples corresponding to a scenario in which the Navigator software (possibly developed by a third-party vendor) misbehaves and gives wrong navigation commands. CRV was also able to precisely attribute the attacks to the misbehavior of the flight navigator. Once contingencies were included (e.g., added redundancy via three navigator software from different vendors) in the system design, CRV was able to prove that the system was resilient with respect to the given properties.

V. Test Generation

VERDICT Automated Test Generation component generates security related test cases and procedures from the behavioral design and execute the tests against the implementation to check the conformance between the behavioral design and implementation.

A. *Terminology*

A *test objective* is a set of input/output constraints that the generated test needs to satisfy. Test objectives can be derived from standards or provided by system designer.

A *test case* is a human readable or machine-readable description of the test objective including input/output constraints, traceability information, purpose of the test, etc. Test cases need to be reviewed either manually or automatically to make sure the set of tests satisfy the test coverage criteria.

A *test procedure* is a sequence of test steps derived from test cases, with each step defining the input values, expected output values, tolerance, traceability information, etc. Test procedures are written in a common machine-readable form without specifying test environment specific information.

A *test script* is a test environment specific executable script that is translated from a test procedure. The test scripts use the language and format required by the test environment and map the input/output variables in test procedures to test interfaces in the test environment.

B. *Automated Test Generation Workflow*

Fig. 5 shows a high-level VERDICT workflow applied to a typical system development process. The left-most column describes a typical system development process where system engineers first perform architecture design, and then refine the architecture design with behavioral descriptions. Software engineers, hardware engineers, etc. will then implement the system based on the refined system design with behavior definitions. The VERDICT tool helps to assure the cyber-resiliency of the system under this development process by: 1) using VERDICT Model-Based Architectural Analysis module to detect architecture level vulnerabilities and suggest security controls to reduce the likelihood of successful attacks, 2) applying VERDICT Cyber-resiliency Verifier module to reveal behavioral level vulnerabilities and suggest placement of runtime monitors considering certain threat effects, and 3) applying VERDICT Automated Test Generation module to generate behavioral level security related test cases and procedures that are executable in an implementation compatible testing environment with the intent to reveal implementation errors or inaccuracies.

Automated Test Generation enables efficient test-based verification of the system design and implementation and complements formal verification to provide a comprehensive assurance package for system cyber-resiliency. In VERDICT, Automated Test Generation is a sub-component of the behavioral analysis module. It gets the inputs from the behavioral analysis module and writes the outputs along with the behavioral analysis. Fig. 6 shows the high-level workflow of the automated test generation component. The component takes inputs of: 1) formal cyber-resiliency properties defining constraints for the system behaviors, 2) threat effect models, and 3) behavioral models describing high level system behaviors. Based on the inputs the component generates three sets of outputs: 1) test cases for engineers to review, 2) machine readable test procedures for engineers to process separately, and 3) executable test scripts targeting specific test environments. Internally, the Automated Test Generation component processes the formal cyber properties and threat models to identify security-critical and error-prone test objectives, and then instruments the identified test objectives in the behavioral models so reasoning engines such as Kind 2 can process and provide input-output sequences that satisfy the test objectives. The identified test objectives are also used to generate human readable test cases by analyzing their test coverage and applying variable or terminology mappings for human readable purposes. The generated input-output sequences from the solver are used to generate machine readable test procedure files describing step-by-step instructions of running the tests, which can then be translated to test scripts that are executable in various test environment.

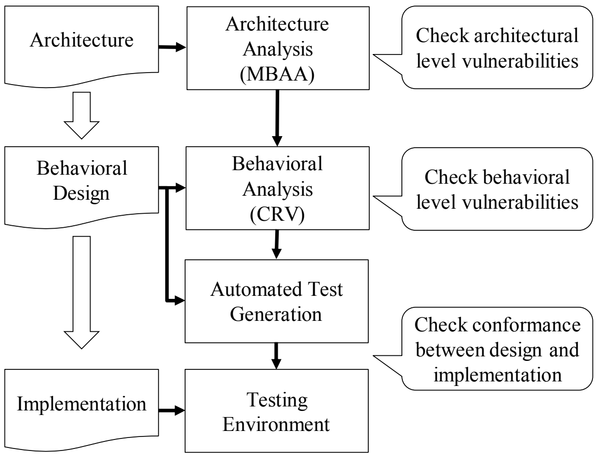
Architecture
Architecture Analysis (MBAA)
Check architectural level vulnerabilities
Behavioral Design
Behavioral Analysis (CRV)
Check behavioral level vulnerabilities
Automated Test Generation
Check conformance between design and implementation
Implementation
Testing Environment

Figure 5. Automated Test Generation in VERDICT Workflow.

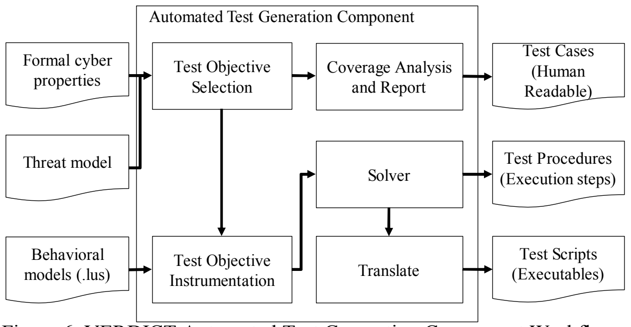
Automated Test Generation Component
Formal cyber properties
Threat model
Behavioral models (.lus)
Test Objective Selection
Coverage Analysis and Report
Solver
Test Objective Instrumentation
Translate
Test Cases (Human Readable)
Test Procedures (Execution steps)
Test Scripts (Executables)

Figure 6. VERDICT Automated Test Generation Component Workflow.

C. *Test Objective Selection*

Test objectives are expected to be selected from both formal cyber-resiliency properties and threat effect models. Tests from formal cyber-resiliency properties are generated and executed to confirm that for the given tests, the system implementation conforms to the same cyber-resiliency properties that the verified design satisfies. Tests from threat effect models, on the other hand, provide confidence that the system implementation behaves the same as the system design under given threat effects.

Consider the following property: In autonomous mode, the UAV takes a picture only when it reaches the waypoint specified in Task 1 (x=1, y=1). The test objective of the property is written as a trap condition: In autonomous mode, the UAV will never take a picture when it reaches the waypoint specified in Task 1. The trap condition is set as a guarantee of the UAV Lustre model and Kind 2 solver is able to generate a counter example for the trap condition which is the test case of the original property.

Consider the following threat effect: GPS channel may deviate from the true value due to threats related to integrity. The test objectives for the threat effects include: exercising maximum deviation to both sides of the value range and exercising minimum deviations around the true value.

D. *Property-based Test Generation for HawkeyeUAV*

A property-based test generation tool has been implemented that takes the properties verified by behavioral analysis and generate one test procedure for each property. This tool generates test procedure report for “UASSystem” which is the top-level component of the HawkeyeUAV system. Fig. 7 shows one of the generated test procedures. The top-left corner of the

Input Name	**Step 0**	**Step 1**	**Step 2**	**Step 3**	**Step 4**
location_source_pos.x	0	0	0	1	1
location_source_pos.y	0	1	1	1	1
location_source_dir	North	North	East	East	East
Output Name	**Step 0**	**Step 1**	**Step 2**	**Step 3**	**Step 4**
cmd	IncreaseY	IncreaseYaw	IncreaseX	NoChange	NoChange
payloadCmdFC	None	None	None	None	Camera

Figure 7. Property-based Test Procedure Generated from VERDICT.

test procedure describes the property where the test procedure is generated from along with the runtime for test generation, status of the test generation, and number of generated test steps. Following that, the test procedure file describes the test inputs and expected outputs for the component under test. For example, Fig. 7 shows the test procedure generated to satisfy property a of the HawkeyeUAV: In autonomous mode, the UAV takes a picture only when it reaches the waypoint specified in Task 1. The test procedure consists of three test inputs, two expected outputs and five test steps, which is intended to be performed on the implemented HawkeyeUAV. The test procedure tests if the implemented system can correctly respond to the GPS inputs and produce commands to drive the UAV to Task 1 target point and take a picture.

VI. CONCLUSION

This paper introduced the Model-Based Architecture Analysis and Cyber-resiliency Verifier functionality that are under development in the VERDICT project on the DARPA CASE program. The Hawkeye UAV mission requirements, cyber-resiliency properties and architecture model provided a simplified but relevant use case for the tool's development. The MBAA function generated a clear and quantified vulnerability metric, an attack-defense tree and a set of design recommendations to influence resiliency. The CRV function demonstrated the ability to capture formal cyber-security properties and abstract behavioral models in a system engineering design tool. The CRV function provided the capability to instrument the abstract design model with threat effects and perform analysis that returns either: 1) formal proof that the design satisfied the cyber-resiliency properties, 2) a counter-example illustrating a failure case including blame assignment, or 3) placement of a runtime monitor that ensures identification of the effect of an attack. The CRV also generated a set of cyber-requirements-based test cases to apply for traditional test-based verification. Phase 1 of the program which ended in February of 2019 demonstrated a proof of the concept and created an initial tool prototype. The team is continuing to develop and mature the tool's capability in Phase 2 of the program that is expected to run through summer of 2020. In this phase of the program the team will integrate safety modeling and analysis into the MBAA and add the capability to synthesize an architecture that meets both safety and security design constraints. The CRV tool will be updated with more sophisticated threat model instrumentation, blame assignment capability and support for automated test generation for more complex systems. The team will focus on integration with OSATE (Open Source AADL Tool Environment) [18] where architectural design models will be captured in AADL and analysis will be performed with the VERDICT tool implemented as a plugin.

ACKNOWLEDGEMENTS

DISTRIBUTION A. Approved for public release: Distribution Unlimited. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and Space and Naval Warfare Systems Center, Pacific (SSC Pacific) under Contract No. N66001-18-C-4006.

REFERENCES

[1] TechTarget Network, "How air gap attacks challenge the notion of secure networks," TechTarget Network, 27 April 2018. [Online]. Available: https://searchsecurity.techtarget.com/essentialguide/How-air-gap-attacks-challenge-the-notion-of-secure-networks. [Accessed 10 July 2019].

[2] D. Brecht, "Pros and cons in penetration testing services: the debate continues," Infosec, 30 November 2016. [Online]. Available: https://resources.infosecinstitute.com/pros-and-cons-in-penetration-testing-services-the-debate-continues/#gref. [Accessed 10 July 2019].

[3] Carnegie Mellon Software Engineering Institute, "Architecture Analysis & Design Language," 2015. [Online]. Available: http://www.aadl.info. [Accessed 10 July 2019].

[4] "PX4 autopilot," [Online]. Available: http://px4.io. [Accessed 11 4 2019].

[5] B. Kordy, S. Mauw, S. Radomirović and P. Schweitzer, "Foundations of attack-defense trees," in *International Workshop on Formal Aspects in Security and Trust*, Berlin, Heidelberg, 2010.

[6] B. Kordy, S. Mauw, S. Radomirović and P. Schweitzer, "Attack–defense trees," *Journal of Logic and Computation, 24(1),* pp. 55-87, 2014.

[7] "DO-356 Airworthiness Security Methods and Considerations," RTCA, 2014.

[8] MITRE, "Common Attack Pattern Enumeration and Classification," [Online]. Available: https://capec.mitre.org/.

[9] National Institute of Standards and Technology (NIST), "NIST Special Publication (SP) 800-53 Revision 5 (Draft), Security and Privacy Controls for Systems and Organizations,," Gaithersburg, Maryland, 2017.

[10] A. Crapo and A. Moitra, "Toward a unified English-like representation of semantic models, data, and graph patterns for subject matter experts," *International Journal of Semantic Computing,* pp. 215-236, 2013.

[11] "Semantic Application Design Language (SADL)," [Online]. Available: http://sadl.sourceforge.net/. [Accessed 8 2 2019].

[12] "OWL: Web Ontology Language," [Online]. Available: https://www.w3.org/OWL/. [Accessed 8 2 2019].

[13] "Xtext: Language Engineering for Everyone!," [Online]. Available: http://www.eclipse.org/Xtext/. [Accessed 8 2 2019].

[14] A. Javaid, W. Sun, V. Devabhaktuni and M. Alam, "Cyber Security Threat Analysis and Modeling of an Unmanned Aerial Vehicle System," in *IEEE Conference on Technology for Homeland Security (HST)*, Waltham, MA, 2012.

[15] P. Manolios, K. Siu, M. Noorman and H. Liao, "A Model-Based Framework for Analyzing the Safety and of System Architectures," in *RAMS*, Orlando, 2019.

[16] D. Dolev and A. C. Yao, "On the security of public key protocols," Stanford Tech. Rep., 1981.

[17] A. Champion, A. Mebsout, C. Sticksel and C. Tinelli, "The Kind 2 model checker," in *International Conference on Computer Aided Verification*, 2016.

[18] "Welcome to OSATE," [Online]. Available: https://osate.org. [Accessed 2019 10 July].