Zero-Trust Design and Assurance Patterns for Cyber-Physical Systems

Saqib Hasan, Isaac Amundson, David Hardin
Collins Aerospace
Email:{saqib.hasan, isaac.amundson, david.hardin}@collins.com

***Abstract*—Security is paramount in all mission-critical domains, including the aerospace industry. Cyber-attacks are increasing both in number and sophistication. Zero-trust is an emerging initiative that has proven very effective for enterprise systems in the Information Technology domain; however, research is lacking on applicable zero-trust mechanisms and their assurance for cyber-physical systems (CPS). We have already identified various zero-trust mechanisms in our previous work. In this paper, we present our zero-trust architecture design patterns and provide a methodology for the assurance of these mechanisms. Towards this objective, we have identified an initial set of assurance patterns covering individual zero-trust components in a system design. Our design and assurance patterns are made available to system engineers in pattern libraries. Engineers can model system architectures and utilize one or more of these patterns to provide design assurance based on individual zero-trust security requirements to improve the overall system cyber-security. To demonstrate our approach, we apply our assurance patterns to an unmanned aerial vehicle surveillance application. We discuss how our framework leverages the use of these patterns to develop zero-trust-enabled systems with different security requirements. Furthermore, our assurance patterns enable engineers to identify any design flaws and correct them during the initial system design phase, thus saving development time, effort, and cost. As a result, the overall approach can be utilized to design system models with specific zero-trust security requirements to improve the security posture of a CPS.**

***Keywords*— Zero-trust, Cyber-physical systems, Design patterns, Assurance patterns, Security, Cyber-attacks, Assurance arguments, Assurance fragments.**

I. Introduction

Security is becoming paramount in almost every domain. It plays an even more important role in mission-critical systems such as aviation systems. Several incidents have been reported recently that demonstrate the ever increasing number of cyber attacks on high-assurance systems [1], [2], [3], [4]. These attacks are not only increasing in number but also in sophistication. For example, a Boeing subsidiary that provides flight planning and navigation tools reported a cyber-attack mentioning flight disruptions [5]. A ransomware-based attack affected the flight operations of the Spicejet airline, resulting in severe flight delays [6]. A similar ransomware attack was reported by a Canadian military contractor where a third-party gained access to their computer network that disrupted their operations [7]. Such scenarios can result in catastrophic consequences and require the need for incorporating security as part of the design considerations for such systems.

Zero-trust is an emerging initiative that has proven very effective in improving the security posture of systems in various domains. We have demonstrated an approach to support this claim in [8]. There are two aspects in addressing this challenge. First, we need the tools and technologies to ensure that engineers have the ability to design systems with zero-trust security considerations in mind. Second, once these systems are implemented, engineers should be confident that the realized system is in fact secure. As a result, providing assurance of the system design is a very critical aspect. Moreover, assurance for zero-trust enabled systems is a challenge in itself as it is currently unexplored. Such assurance can also help streamline the certification effort, thereby reducing the time and cost involved. In order to achieve this, assurance patterns can be leveraged to automatically construct assurance arguments based on a system's security specification.

Several researchers have demonstrated the use of assurance patterns for both safety and security. Work published in [9] describes a new approach to construct safety arguments where one argument represents the confidence of another. Authors in [10] focus on categorizing safety patterns and reusing them appropriately as manual construction of safety arguments can be very time consuming. Researchers in [11] discuss a modular approach in the construction of argument patterns and argue their benefits when dealing with large safety cases. In the past, system assurance has proved their importance in traditional CPS and as a result work is being done to identify their application for assurance of systems equipped with newer technologies such as the use of learning-enabled components. Research published in [12] demonstrates a methodology for the assurance of machine learning in autonomous systems (AMLAS). The work describes the systematic integration of safety assurance as part of machine learning component development and generating required evidence to support the argument.

Further, work discussed in [13] focuses on an approach that documents an assurance case specifically for system security. Other research published in [14] discuss argument patterns and their use for both safety and security cases. The authors have proposed an approach based on automated instantiation of these patterns using a model-based instantiation process. A recent work [15] discusses the role of design patterns in providing solutions to common recurring design problems. In addition, the authors discuss the importance of assurance for the design patterns. However, none of the works discuss assurance patterns targeted specifically to CPS zero-trust architectures.

Considering the above technological gaps this paper provides the following contributions:

- We present an initial set of design patterns specifically developed to support individual zero-trust components in a system design.
- We have designed corresponding assurance patterns for the individual zero-trust design patterns, enabling the linking of a system model and its assurance together. The mechanism provides guidance to engineers if the design violates any of the zero-trust requirements.
- The assurance patterns are implemented as individual fragments that are made available in a pattern library to system engineers. Engineers can model systems in a modeling language such as AADL and utilize one or more of these patterns to provide design assurance based on individual zero-trust requirements to improve the overall system security.
- To demonstrate our approach, we apply our assurance patterns to an unmanned aerial vehicle (UAV) surveillance application. We discuss how our framework leverages the use of these assurance patterns from a pattern library to develop zero-trust enabled systems with different security requirements. Furthermore, our assurance patterns enable engineers to identify design flaws and correct them during the initial system design phase, thus saving development time, effort, and cost.

The rest of the paper is organized as follows. First, in Section II, we provide a background on the concepts that are discussed in the paper, including the zero trust principle, system architecture modeling in AADL, assurance patterns and arguments, and their representations. Next, in Section III, we discuss zero-trust design and associated assurance patterns. We provide an example of our approach using an experimental UAV model in Section IV. Finally, in Section V, we provide conclusions and discuss future work.

II. Background

Before we dive into our zero-trust approach, it is important to describe some of the terms and concepts we will be using in this paper. In this section, we first discuss the concept of zero-trust and its tenets. Next, we discuss assurance arguments and patterns, along with their notations. We then discuss the architecture modeling language used for system design. Finally, we provide an overview of our assurance tool for evaluating system models against assurance goals.

A. Zero Trust:

Zero trust is an emerging initiative which is collaboratively explored by joint efforts between the National Security Agency (NSA), DoD CIO, US Cyber Command (USCYBERCOM), and DISA. According to [16], "Zero trust is a cybersecurity paradigm focused on resource protection and the premise that trust is never granted implicitly but must be continually evaluated". Zero trust focuses on moving from a traditional perimeter based infrastructure to a perimeter-less design. To achieve this, zero trust relies on basic core tenets, namely:

- **Presume a breach.** The presence of an adversary is assumed within the operational environment at all times.
- **Never trust, always verify.** Any access to a resource is only granted after explicitly authenticating the devices/requests and the access decision is valid only for a specific request.
- **Assume a hostile environment.** Every entity, such as devices and networks, that constitutes the operational environment is considered untrusted.
- **Apply unified analytics.** Utilize state-of-the-art analytical capabilities such as machine learning and deep learning algorithms to support and improve access control policies.
- **Scrutinize explicitly.** Access to resources is always conditional and the policies associated with access decisions are dynamic in nature. These policies depend on the confidence level and actions of the requester. Whenever user action results in a change in the confidence level, the access policies should be changed dynamically.

These tenets are described in detail in [16] and [17].

Figure 1 demonstrates the concept of a simple zero trust architecture. The subjects (for instance, devices) are on the left hand side and a resource (for instance, a database) is on the right. A policy decision point/policy enforcement point (PDP/PEP) sits between the two in the middle. The zone to the left of PDP/PEP is implicitly considered untrusted while the zone to its right is a trusted zone. Whenever a subject requests access to a resource, the PDP/PEP evaluates the request and grants access to the resource only if the trust between the PDP/PEP and the subject is validated. The PEP/PDP has the ability to use information from various systems such as Security Information and Event Management (SIEM), Data Rights Management (DRM), etc., to further employ more rigorous policy checks that can then be utilized for access decisions.

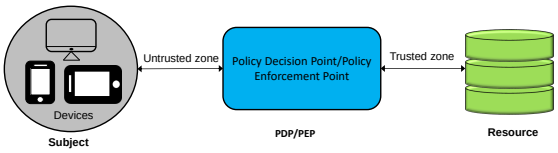

Fig. 1. Zero trust architecture concept.

B. Assurance Arguments and Assurance Patterns:

Assurance *arguments* are defined as a compelling argument satisfying a property which is supported using a body of evidence. Assurance *patterns* on the other hand are defined as reusable generic arguments (preferably defined through consensus by subject matter experts) that can be instantiated with a system instance to produce a concrete assurance argument. Both assurance patterns and assurance arguments can be represented using various notations such as Claims, Arguments

and Evidence (CAE) [18], Justification Diagrams [19], and Goal Structuring Notation (GSN) [20]. In this paper, we utilize GSN to represent our assurance patterns. We demonstrate the use of our assurance patterns for generating concrete assurance arguments in Section IV. Here, we describe GSN in a bit more detail for the sake of clarity. GSN is a means for representing an argument (typically a dependability argument) in a structured manner. It uses several basic elements to construct the assurance argument. These elements are shown in Figure 2. The key elements we use are defined as follows:

- **Goal:** A goal represents a claim that forms the assurance argument. It is symbolized by a rectangle.
- **Solution:** A solution represents the evidence that supports a goal. It is symbolized by a circle.
- **Strategy:** A strategy represents the reasoning steps as to how a goal is supported by sub-goals or solutions. It is symbolized by a parallelogram.
- **Context:** Describes the context in which a goal or strategy should be interpreted. It is symbolized by a rectangle with rounded corners.
- **Assumption:** Describes the assumptions associated with a goal or strategy and is symbolized by an oval with the letter 'A'.
- **Justification:** This element describes the justification for using a specific goal or strategy. It is symbolized by an oval with the letter 'J'.
- **Undeveloped:** An undeveloped element when connected to a goal or strategy indicates that the remainder of the argument is not yet specified. It is symbolized by a diamond.
- **Uninstantiated:** An uninstantiated element when connected to a GSN element indicates the attached element has not yet been instantiated with a system context. It is symbolized by a triangle.

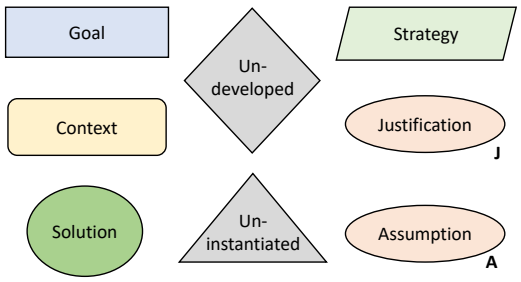

Fig. 2. GSN elements.

The above elements are connected together to form assurance patterns/arguments for covering a variety of dependability properties. GSN is described in further detail in [20].

C. Architecture Analysis and Design Language (AADL):

AADL is an architecture language used for modeling real-time distributed embedded systems [21]. It enables the user to capture important and realistic design concepts and hence it is very well suited to design models for the aerospace industry. Another aspect of the AADL language is that it provides the ability to hierarchically capture both software and hardware related details. Some of the hardware components represented in the AADL language include buses, devices, processors, and memory. Software components include threads, processes, and data. AADL also provides users the capability to specify connections, interfaces, properties, and data flows. In addition, AADL suits perfectly for designing systems whose models are created incrementally and refined over time, thereby providing a high degree of design flexibility. One of the more useful features of AADL compared with other modeling languages is that it includes an *annex* mechanism that enables the language to be extended with domain-specific features. AADL provides a reference framework called the Open Source AADL Tool Environment (OSATE) [22].

AADL provides the following key modeling constructs:

- **AADL components:** Figure 3 represents the AADL components that are utilized in this paper as part of our design patterns and also the modeling in our UAV use case. Figure 3 illustrates the modeling language component hierarchy. A system in AADL is represented using the rounded rectangle. A *system* represents the outermost abstraction of a model. A system can contain one or more components, including other systems. A *process* component, represented using the parallelogram, is used as an address space in the model whereas a *thread*, shown by the dashed parallelogram, denotes a schedulable execution flow. A process must contain at least one thread. A process (or thread) can be bound to a *processor*, which is represented using a 3D parallelogram. To represent data storage or *memory*, a cylinder is utilized. A *device* represents a black box system. Finally, connections between components are modeled using solid arrows that indicate directionality of data flow. There are other elements that are part of the AADL language but they are not referenced in this paper.

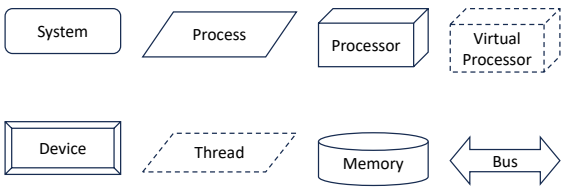

Fig. 3. AADL components types.

- **AADL properties:** We have utilized AADL properties as part of our modeling process. A property in AADL is a typed attribute that can be associated with one or more components. It follows a specific syntax and is assigned as a *property association*. Some of the allowed

types of properties in AADL are aadlboolean, aadlinteger, aadlreal, aadlstring, enumeration, classifier, reference, and list. A property can be inherited by a subcomponent of a parent component.

- **AADL Component Connections:** A component connection in an AADL model defines the interactions, control flow and data flow between various components in the model. Each component contains details of their internal connections. A feature represents the interface of a component. It has a feature name, a category, and a direction. A feature category specifies the type of component interaction with other model components. These categories can be defined using different types of ports such as event port, data port, and event data port. The port direction can be defined as part of the feature direction definition (input, output or both).

We used AADL to model our design patterns and example models. These elements and other details about the AADL language are described in detail in [21].

D. Resolute:

Resolute [23] is an assurance case language and tool created as an annex in AADL. It is well suited for defining assurance patterns since it provides the ability to link both a system architecture and its assurance together. Users can create claims and rules to satisfy these claims. Each of these claims and rules can be parameterized by elements from the model. Furthermore, the rules and claims are parameterized by specific types as defined in the model. When Resolute uses a specific system model to generate a concrete assurance case from an assurance pattern, this process is referred to as *instantiation*. Resolute provides various built-in functions to query the model for determining whether the architecture supports specific claims. It also allows the ability to query external artifacts for any evidence that is needed to support these claims by calling user-defined analysis plugins and function libraries. Resolute follows the GSN standard to allow users to construct assurance patterns compatible with the GSN format.

Figure 4 represents an example assurance pattern fragment demonstrating the mapping between Resolute and GSN elements. On the left-hand side is the representation of an assurance pattern in Resolute following the GSN notation and on the right-hand side are the actual GSN element representations. Each of the GSN elements utilized in our assurance pattern is color coded with respect to its corresponding representation in Resolute.

Figure 5 shows the system assurance process using our Resolute framework. The input to this framework is a system model developed in AADL. Resolute provides an editor to define assurance patterns following the GSN standard as described above. The patterns can then be stored in an assurance pattern library. Pre-existing patterns can be directly utilized within the Resolute environment for use in providing assurance for various systems. Resolute is capable of querying the AADL system model, which is needed during the assurance evaluation phase. Whenever the user wants to check the assurance

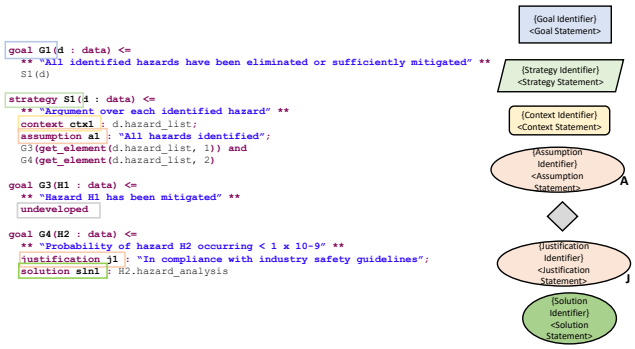

Fig. 4. Example assurance pattern representation in Resolute following the GSN standard.

corresponding to a specific system model, the Resolute engine triggers the Resolute evaluator to perform the task. The input to the Resolute evaluator is a specific assurance pattern and the system context (i.e., the AADL system or subsystem model). This process is referred to as assurance pattern instantiation in our paper.

Resolute then evaluates the entire pattern using the rules specified within each goal that describe the evidence needed to satisfy them. As a result, an assurance case is generated as an output of this process. This assurance case can be a passing or a failing argument depending upon the analysis performed by Resolute. If all the required evidence is successfully collected, then it results in a passing assurance case, otherwise Resolute will generate a failing assurance case and highlight the goals that are missing evidence for substantiation. Please note that we have demonstrated our approach of generating assurance cases by leveraging our Resolute framework, but the patterns defined in the following section are not framework dependent but are rather generic patterns.

III. Zero-Trust Design and Assurance Patterns

In this section, next, we present each of our zero-trust design patterns along with their corresponding assurance pattern.

Zero trust design patterns provide engineers with templates to model systems with built-in zero-trust security mechanisms for improving overall system security. These design patterns can be utilized in system architecture models based on various factors such as the number of critical resources that need to be protected, level of security that the system requires, maintenance cost of the system, and the overall budget available to protect the system. In addition, once the architecture model includes these design patterns, it is necessary to ensure that the system is in fact zero-trust enabled per the requirements. This can be achieved by leveraging a corresponding assurance pattern for each zero-trust design pattern. Each of our design patterns satisfy one or more of the zero-trust tenets listed in Section II (however, this work does not address the *unified analytics* tenet).

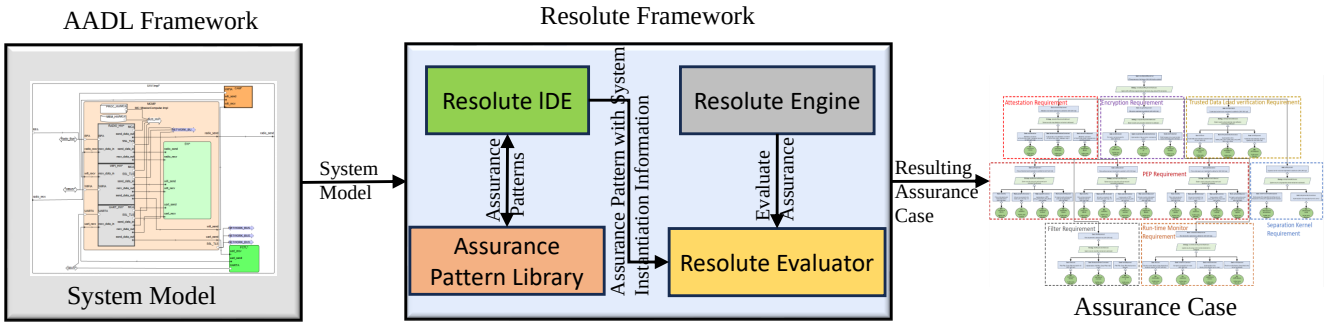

Fig. 5. Assurance using Resolute.

A. Secure Data Load

Secure data load consists of two components, *encryption* and *decryption*. It ensures that the integrity and authenticity of system data is maintained while in transit and at rest.

1) Encryption: Encryption [24], [25] ensures data integrity and authenticity by encrypting data at the source. The encryption mechanism is shown in Figure 6. It consists of an *Encryption_Manager* and an *Encryption_Policy* component. The rules for encrypting the data are defined in the *Encryption_Policy* component as the security policy. Whenever input data is received at its port, the *Encryption_Policy* causes the *Encryption_Manager* to encrypt the data by utilizing the *Cryptographic_Algorithm* and *Cryptographic_Keys*. It then sends the encrypted data back to the *Encryption_Policy* component, which passes it as an output.

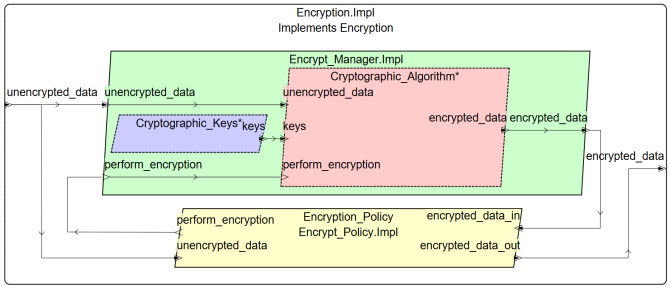

Fig. 6. Architecture for encryption.

Next, we provide the design assurance of the encryption mechanism to ensure that the pattern is applied correctly. Figure 7 shows the assurance pattern fragment for the *Encryption* mechanism. The top level claim '*encryptionCompliance*' argues that encryption is supported for the given system. It is associated with a strategy '*encryptionComplianceAddressed*' that is supported by three sub-goals. Each of the sub-goals is discussed further in detail:

- **existsEncryption:** This claims the *Encryption* mechanism exists in the system model. Specifically, this goal ensures that each communication bus enforces an encryption protocol, making sure that all messages passing between components are encrypted. Intuitively, the overall claim ensures if any physical bus is not bound to an encryption bus then it clearly violates our objective of ensuring data/message encryption within the zero-trust enabled system. In our assurance pattern, this goal is supported by the solution '*chkEncryption*', which describes the required encryption protocol binding.
- **encryptionImplementationCorrectness:** This goal ensures that the implementation of the encryption mechanism is correct. In our approach, this type of goal can be supported using test results for verifing the implementation. In our pattern, this claim is supported by the solution node '*chkEncImpCorrectness*'.
- **dataPortBounded:** This claim ensures that message ports of all the drivers within the zero-trust system model are bounded to an encryption bus. Although we have already ensured that individual physical buses are bound to the encryption buses, this claim further ensures that each data port that requires encryption is properly bound to the correct encryption bus. In our pattern, this claim is supported by the solution node '*chkPortBounded*', which specifies the necessary bindings for the bus architecture.

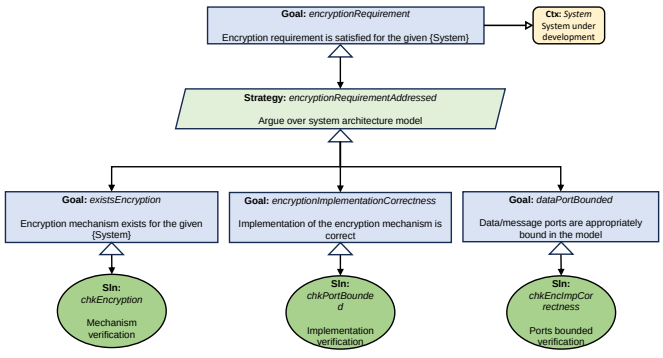

Fig. 7. Assurance pattern for encryption.

2) Decryption: The decryption mechanism [25], [26] is represented in Figure 8. Decryption securely decrypts the data from its original source at its destination. The architecture of decryption is very similar to the encryption mechanism. The only difference is that instead of *Encryption_Manager*

and *Encryption_Policy* components it consists of a *Decryption_Manager* and a *Decryption_Policy* component. Whenever encrypted data appears at the input port of the decryption mechanism, the data is decrypted using *Cryptographic_Algorithm* and *Cryptographic_Keys* present within the *Decryption_Manager*. The decrypted data is then passed to its output port via the *Decryption_Policy* component.

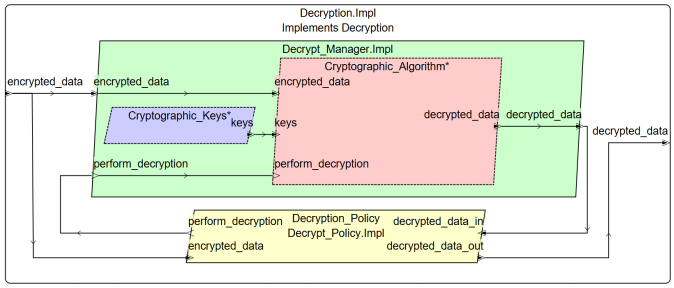

Fig. 8. Architecture of the decryption mechanism.

The design assurance for decryption is similar to encryption and hence it is not discussed here.

B. Attestation

Figure 9 represents the architecture pattern for the attestation component. Attestation [27], [28] ensures software authenticity and integrity. The pattern consists of two subcomponents, *Measured_Boot* and *Secure_Boot*, to perform attestation. The *Attestation_Policy* defines the rule for successful attestation. The input to both *Secure_Boot* and *Measured_Boot* is a *root_of_trust*. Both these mechanisms utilize the *root_of_trust* to generate individual data structures such as *quote* and *validation*. The *validation* represents information regarding software integrity whereas *quote* contains detailed information about the software such as its version, configuration details, etc. The *validation* and *quote* messages are further utilized by the *Attestation_Policy* component to provide the attestation decision.

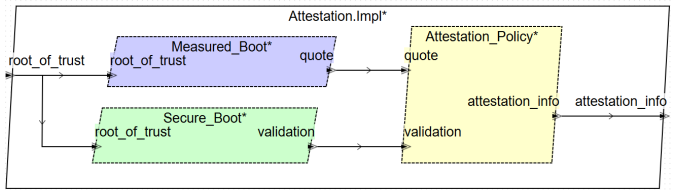

Fig. 9. Architecture of the attestation mechanism.

Next, we provide the design assurance of the attestation mechanism to ensure that the desgin pattern is applied correctly. Figure 10 shows the assurance pattern fragment for the *Attestation* mechanism. The top-level goal '*attestationRequirement*' ensures that the compliance for attestation is achieved for the given system. We substantiate the claim using strategy '*attestationRequirementAddressed*' in which we argue over the system architecture model. This strategy is further supported by three sub-goals. Each of these are discussed further in detail:

- **existsAttestation:** This goal identifies the presence of the *Attestation* mechanism in the system model. In addition, the goal ensures that its connections are connected appropriately to other components in the system. Please note that the component to which the attestation component needs to be connected is passed as an input to this goal. The goal is supported by the solution node '*chkComponentExists*', which specifies the component instances the model must contain for compliance.
- **attestationImplementationCorrectness:** This claim ensures that the implementation of the attestation mechanism is correct. It is supported by the solution node '*chkAttestImpCorrectness*' in our assurance pattern. This goal is supported by a solution that specifies the implementation correctness evidence such as verification results.
- **attestationNotBypassed:** By bypassing, we mean specific connection(s) of a component are not receiving information from a source component without passing through the required attestation components first. This goal guarantees that specific ports of the attestation mechanism are not bypassed. If the attestation is bypassed it could result in severe security vulnerabilities. Hence, it is important to capture such design flaws that could violate the requirement. The ports of interest can be specified as input to this goal during pattern instantiation. In our pattern, we focus on the '*attestation_info*' port to ensure it is not bypassed. *Attestation_info*' carries critical information that is utilized by components such as the policy enforcement point (see below) to make a trust validation decision for access to a resource. Therefore, this goal is supported by the solution node '*chkAttestationBypass*' in our assurance pattern. The solution specifies evidence that the required connections are not bypassed in the zero-trust system design.

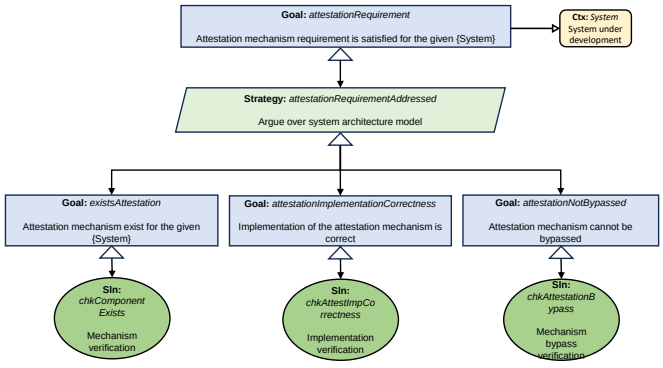

Fig. 10. Assurance pattern for attestation.

C. Policy Enforcement Point

The policy enforcement point (PEP) [29] is represented in Figure 11. The primary function of a PEP is to provide secure

access to resources after explicitly validating trust. A PEP consists of a *Policy_Decision_Point* (PDP) and a *PEP_Manager*. Once an access request is made on the PEP's input, the *PEP_Manager* forwards it to the PDP. The communication interface between the *PEP_Manager* and the PDP is provided by the *Policy_Administrator*. *Policy_Engine* evaluates this request by validating trust using the rules specified within the *Policy_Enforcement_Point_Policy*. Additional information available on the *additional_info* port can also be utilized to validate trust. This port can contain analytical information, specific device-related information, and other platform information. Upon successful validation of trust, the PEP provides access to the requested resource; otherwise access is denied.

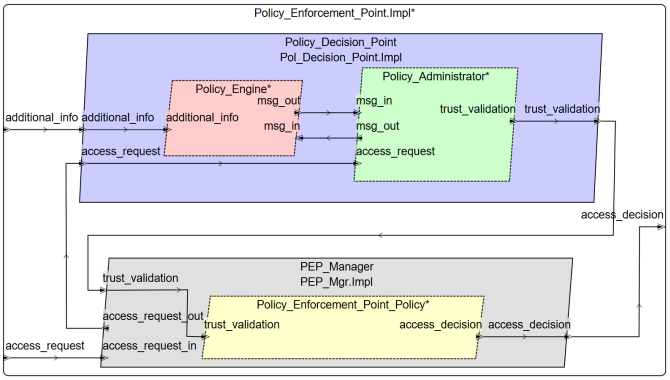

Fig. 11. Architecture of the policy enforcement point mechanism.

Next, we provide the design assurance for the policy enforcement point to ensure that the pattern created is applied correctly. Figure 12 shows the assurance pattern fragment for the *Policy Enforcement Point* mechanism. The top-level goal '*pepRequirement*' ensures that the policy enforcement point compliance is achieved for the given system. We substantiate the claim with the strategy '*pepRequirementAddressed*' in which we argue over the system architecture model. The strategy is supported by four sub-goals. Each of these sub-goals are discussed further in detail:

- **existsPep:** This goal specifies that the *Policy Enforcement Point* mechanism exists in the system model. The goal is supported by the solution node '*chkComponentExists*', which verifies the existence of the given policy enforcement point in the model.
- **pepImplCorrectness:** This goal ensures that the implementation of the PEP mechanism is correct. If the implementation of the PEP is incorrect then the entire zero-trust system requirement can be violated as the correctness of the decision made by the PEP cannot be verified. In our pattern, the solution node '*chkPepImpCorrectness*' ensures this by collecting the required evidence.
- **pepInfoPortConnected:** This goal guarantees: 1) the information port of the given PEP mechanism is connected, and 2) it is connected to the correct component(s) in the system model. If this port is not appropriately connected, the PEP will not be able to make correct access decisions due to a lack of necessary information. This again will result in a system with security vulnerabilities, which attackers can easily take advantage of. The component to which the information port is connected can be passed as an input to this pattern during instantiation. In most cases this port is connected to the attestation mechanism but it can also be connected to any other component that can provide input to the PEP to perform trust validation decisions. In our assurance pattern for the PEP, we further validate this connection by ensuring its association with the appropriate component in the model using solution node '*chkInfoPort*'.
- **pepInputNotBypassed:** This goal ensures that the input ports of the PEP are not bypassed. If the input ports are bypassed then it violates the zero-trust principles mentioned above in Section II, meaning the information from the subject to resources can flow without any mediation. In a zero-trust model this situation can violate the system security requirements. In order to prevent such vulnerabilities due to design flaws, it is necessary to verify the architecture of our system. Hence, in our assurance pattern, the solution node '*chkPepInpPortBypass*' specifies evidence needed to support the goal to prevent such vulnerabilities.

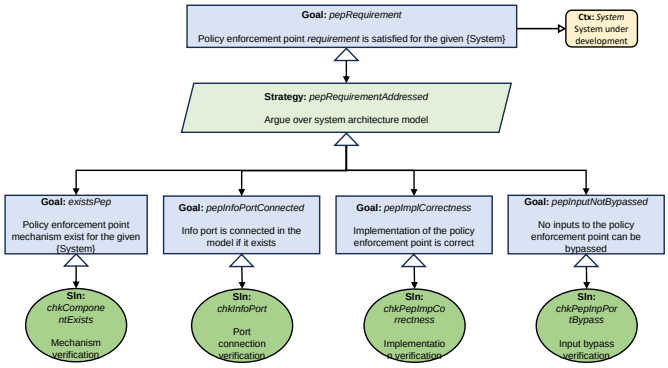

Fig. 12. Assurance pattern for the policy enforcement point.

D. Run-Time Integrity Monitor

The architecture of the run-time integrity monitor [30] is shown in Figure 13. The purpose of the run-time integrity monitor is to identify and flag any abnormal system behavior. The input to the monitor is an observation signal, which could be raw data from a sensor or it could be data from another component. The observation signal can be further processed by a *Signal_Processing_Algorithm* within the monitor. The security policy contained within the *Monitor_Policy* is then utilized to compare the observation signal with a reference value *ref*. The monitor generates an alert if the observation value deviates by more than an acceptable threshold value; otherwise no alert is generated, meaning the system is operating normally. Note that the reference can be passed as an external input to the monitor rather than defined within the monitor itself.

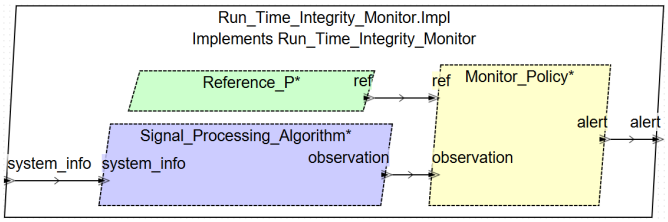

Fig. 13. Architecture of the run-time integrity monitor mechanism.

Next, we provide the design assurance for the run-time integrity monitor to ensure that the design pattern is correctly applied. Figure 14 shows the assurance pattern for the *Run-Time Integrity Monitor*. The top-level goal '*runtimeMonitor-Requirement*' ensures that the system satisfies the zero-trust requirements of this mechanism. It is supported by a strategy '*runtimeMonitorRequirementAddressed*' for arguing over the system architecture model. This strategy is further supported by four sub-goals. Next, we discuss each of these sub-goals in detail:

- **rtmExists:** This goal verifies that the given run-time integrity monitor mechanism exists in the system model. Absence of this mechanism can violate the security requirements for the system. This is supported by the solution node '*chkCompExist*' in Figure 14, which specifies the existence of the run-time monitor component within the system architecture model.
- **rtmImplCorrectness:** This goal ensures that the implementation of the run-time integrity monitor mechanism is correct by collecting the evidence specified in the solution node '*chkRtmImpCorrectness*', as shown in Figure 14.
- **rtmNotBypassed:** Bypassing the run-time monitor will defeat the purpose of including it in the system design. Hence, it is necessary to ensure that the system design is free from such flaws. This goal specifies that the run-time integrity monitor's ports cannot be bypassed. In our assurance pattern, we focus on the '*Alert*' port of the run-time monitor since bypassing this port will result in security vulnerabilities and prevent appropriate actions in case of faults resulting in system failure. Moreover, there could be other ports that could carry useful information depending on the design of the monitor. These ports would again need to be verified to ensure that they are not bypassed. This sub-goal is supported by the solution node '*chkRtmBypass*'.
- **rtmAlertPortConnected:** Another situation could arise where the alert port is not connected, which could make the presence of run-time monitors useless in a system design. Therefore, this goal specifies that the alert port for the run-time monitor is connected. Moreover, merely connecting the alert port doesn't guarantee the correct system design. It is necessary to ensure that the monitor is connected to the appropriate component in the model that utilizes the input from the monitor. Hence, our assurance pattern further verifies this connection by ensuring its association with the respective component in the model. Solution node '*chkAlertPort*' is utilized to collect this information as evidence and support the goal.

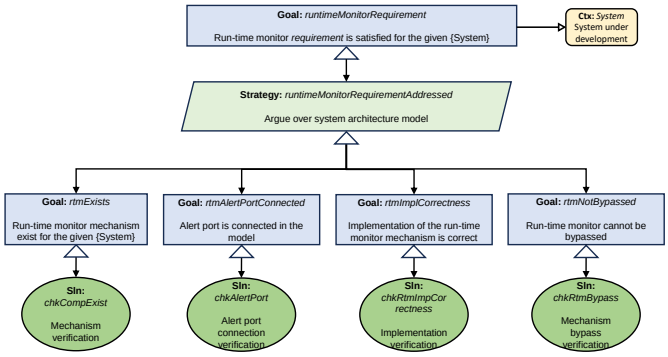

Fig. 14. Assurance pattern for the run-time integrity monitor mechanism.

E. Trusted Data Load

Trusted data load [31] consists of two components, a *Trusted Data Load Verification* process and a *Trusted Data Load Signing* process, ensuring that data load integrity and authenticity is maintained. We describe both of these components in detail.

1) Trusted Data Load Verification: The *Trusted Data Load Verification* process is shown in Figure 15. Signed software or data is the input to *Verification_Process* that needs to be transferred to the target system. The *Data_Load_Verif_Manager* uses this information and utilizes the *Signing_Certificate* and the *Private_Public_Keys* along with the *Verification_Process* algorithm to determine if the information is genuine and originated from a valid entity. To achieve this, trusted data load verification uses the policies set in the *Data_Load_Verification_Policy* and ensures the delivery or installation of data or verified software to the target system.

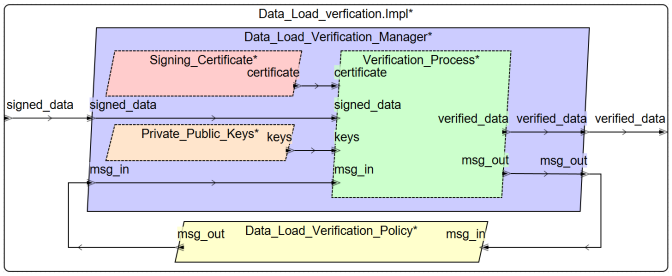

Fig. 15. Architecture of the trusted data load verification mechanism.

Now, we provide the design assurance pattern for the trusted data load verification mechanism to ensure that the design pattern is correctly applied. Figure 16 shows the assurance pattern for the trusted data load verification mechanism. The top-level goal '*tdlvRequirement*' ensures that the trusted data load verification compliance is achieved for the given system. It is associated with a strategy '*tdlvRequirementAddressed*' for arguing over the system architecture model. This strategy

is supported by three sub-goals. Each of the sub-goals is discussed further in detail:

- **tdlvExists:** This goal verifies: 1) the existence of the trusted data load verification mechanism in the model. As absence of this mechanism can violate the security requirements for the system, and 2) the trusted data load verification component is connected appropriately with the rest of the system model. The absence of which can again give rise to security vulnerabilities. Hence, the design of our pattern supports collecting information about the trusted data load component, which determines the evidence for the above two criteria. In our assurance pattern, this is achieved via the solution node '*chkCompExist*'. Besides the existence of the mechanism it argues that it is connected to the correct components such as a PEP and software update module in the system design. In addition, it argues that the source and destination ports are connected appropriately in the architecture model. The required information is passed as in input to the pattern itself during instantiation.
- **tdlvImplementationCorrectness:** This goal argues that the trusted data load verification mechanism is implemented correctly. This claim is supported using the solution node '*chkTdlvImpCorrectness*'.
- **tdlvNotBypassed:** If the output from the trusted data load verification component is bypassed then the purpose of the trusted data load verification component is defeated. Hence, it is necessary to ensure such violations are avoided during design time. This goal guarantees that the trusted data load verification mechanism output port cannot be bypassed. It is supported using the solution node '*chkTdlvBypass*' in the assurance pattern.

2) Trusted Data Load Signing: The architecture for the *Trusted Data Load Signing* component is shown in Figure 17. It is used to sign the software or data that needs to be transported. *Trusted Data Load Signing* ensures that the integrity and authenticity of the software or data is maintained at the source of origination. It is similar to the *Trusted Data Load Verification* process; the only difference is that instead of a *Verification_Process* algorithm, it utilizes a *Signing_Process* algorithm. Whenever *Unsigned_Data* becomes available at the *Signing_Process* component of the *Data_Load_Sign_Manager*, it uses the same *Signing_Certificate* and the same *Private_Public_Keys* to generate a *signed_data*, which ensures that the data or software is signed by a trusted entity.

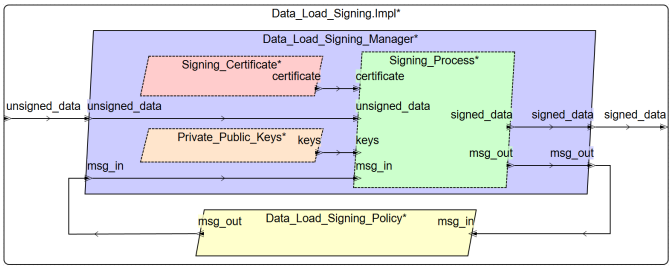

Fig. 17. Architecture of the trusted data load signing mechanism.

The design assurance of the trusted data load signing mechanism is similar to the trusted data load verification mechanism design assurance and hence it is not discussed here.

F. Separation Kernel

The architecture for the separation kernel [32] is represented in Figure 18. A separation kernel ensures integrity of the applications by enforcing space and time partitioning. It adheres

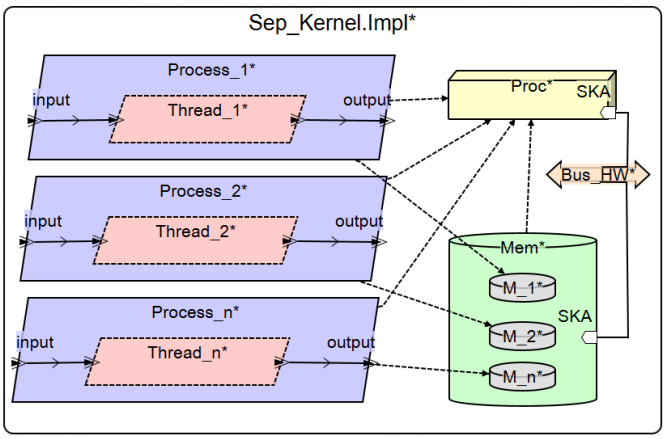

Fig. 18. Architecture of the separation kernel mechanism.

to several zero-trust mechanisms inherently and therefore it is a unique mechanism in itself. The processor (*Proc*) hosts an operating system that provides the separation guarantees and each process (*Process_i*) is bound to the processor. Each process is bound to a specific address space in memory (*Mem*) and contains a single thread representing partitioning in both space and time. Whenever an application executes, it runs within its own process in an isolated address space. Similarly, multiple applications can execute simultaneously within their own processes and memory space, ensuring time and space partitioning thereby maintaining application integrity. Note that although an ARINC 653 annex has been implemented for

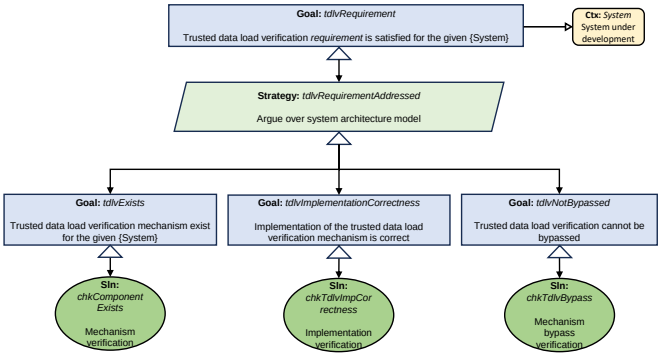

Fig. 16. Assurance pattern for trusted data load verification.

AADL, we do not rely on it here in order to make this pattern more generalizable to other architecture definition languages.

Now, we provide the design assurance of the separation kernel mechanism to ensure that the design pattern is applied correctly. Figure 19 shows the assurance pattern fragment for the separation kernel mechanism. The top-level goal '*skRequirement*' ensures that the separation kernel requirements are satisfied for the given system. It is associated with strategy '*skRequirementAddressed*' for arguing over the system architecture model. This strategy is supported by two sub-goals. Each of these sub-goals are discussed further in detail:

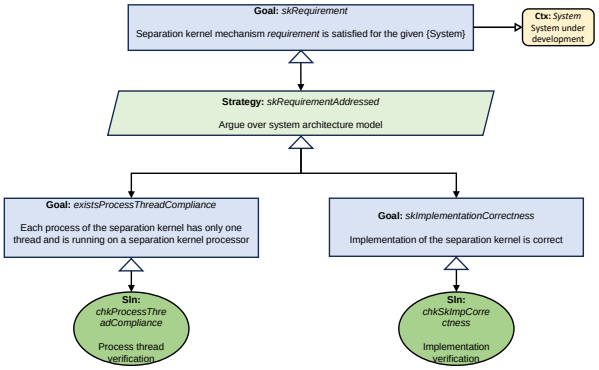

Fig. 19. Assurance pattern for the separation kernel.

- **existsProcessThreadCompliance:** This goal ensures the existence of the separation kernel mechanism in a system model by verifying whether process-thread compliance is met or not. For a system, process-thread compliance simply means that all of its sub-components can have only one thread per process; otherwise the requirement is violated. It is important to capture this design requirement, violation of which could result in a system design where applications do not adhere to running in its own memory space. It is captured using the solution node '*chkProcessThreadCompliance*'.
- **skImplementationCorrectness:** This claim ensures that the separation kernel mechanism is implemented correctly. We capture this using our solution node '*chkSkImpCorrectness*'.

G. Filter

The design pattern for the filter mechanism is shown in Figure 20. A filter component allows only zero-trust compliant inputs to pass through. Compliance with respect to the filter component is defined as a security policy inside the *Filter_Policy* component. When an input arrives on the input port of the filter component, its compliance is verified using the *Filter_Policy*. This compliance check is performed by a *Filter_Algorithm* in addition to the *Filter_Policy* component. If a successful compliance is achieved then the input data is placed on its output port; otherwise the input data is dropped.

Next, we provide the design assurance for the filter mechanism to ensure that the design pattern is applied correctly.

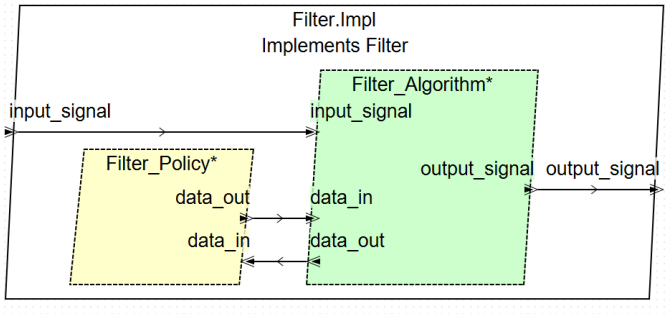

Fig. 20. Architecture of the filter mechanism.

Figure 21 shows the assurance pattern for the filter. The top-level goal '*filterRequirement*' ensures that the filter requirements are satisfied for the given system. It is supported by strategy '*filterRequirementAddressed*' that argues over the system architecture model. This strategy is further supported by three sub-goals. Each of these sub-goals are discussed further in detail:

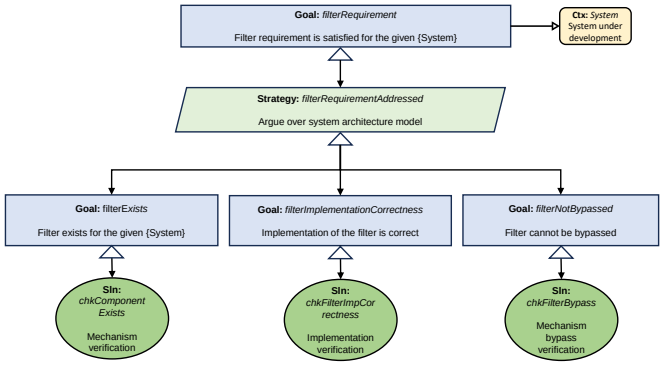

Fig. 21. Assurance pattern for filter mechanism.

- **filterExists:** This goal argues that the filter mechanism is actually present in the model. It utilizes the solution node '*chkComponentExists*' that specifies the required evidence needed for satisfying the sub-goal.
- **filterImplementationCorrectness:** Incorrect implementation of a given filter component can cause the existence of the filter mechanism to be useless. Hence, it is necessary to ensure that the implementation is correct. This goal ensures implementation correctness by using the solution node '*chkFilterImpCorrectness*', as shown in Figure 21.
- **filterNotBypassed:** If the above two sub-goals are satisfied and the filter mechanism is bypassed then the requirements of our secure filter design is violated, which would further violate the overall zero-trust system requirement. Hence, it is essential to identify such critical design issues. Therefore, this goal argues that the filter mechanism cannot be bypassed. For instance, it ensures that the information carried by the input port cannot be bypassed since this situation will violate the requirements of a

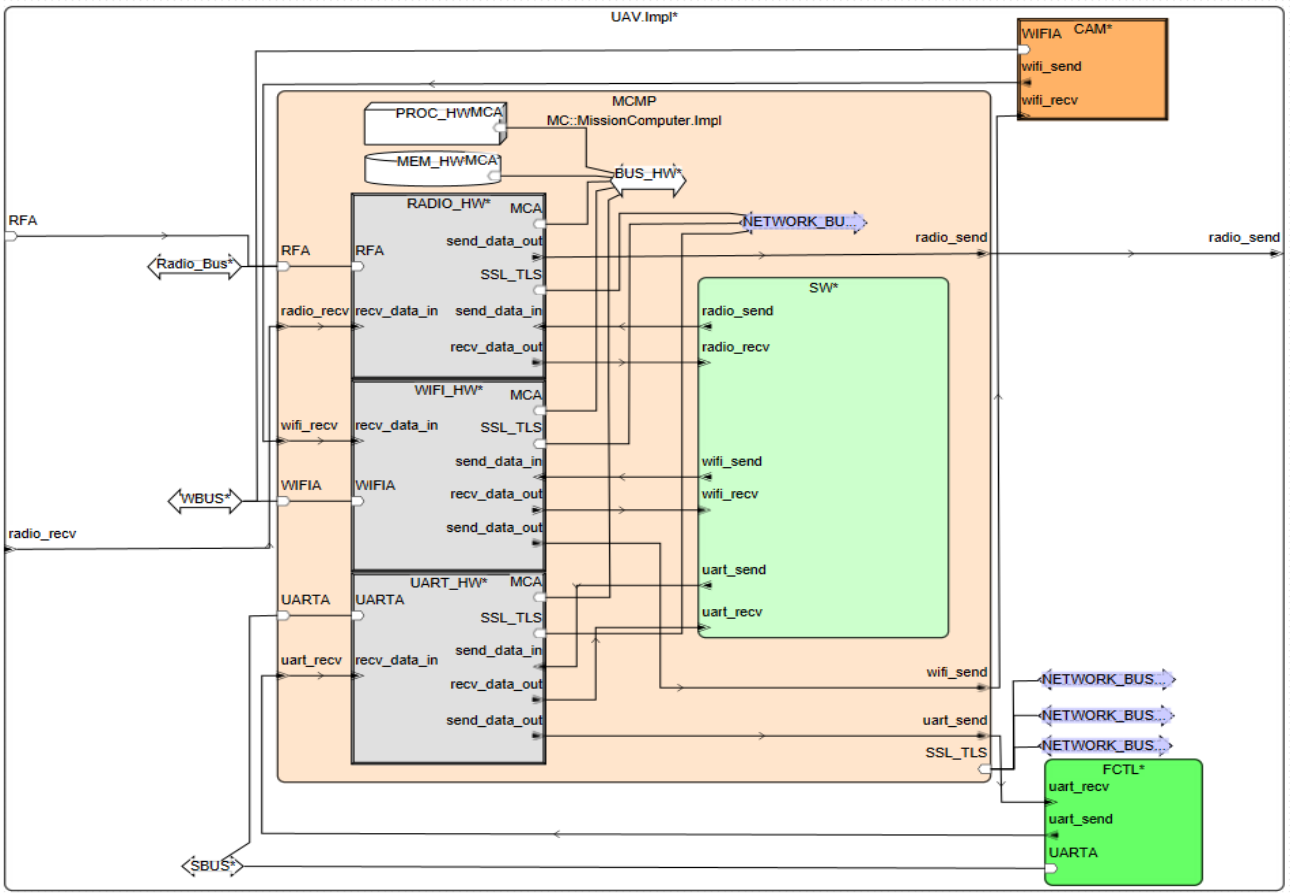

Fig. 22. UAV architecture model in AADL.

filter component and give rise to security vulnerabilities. We have designed the pattern in a way where the user can specify which compnent is responsible to provide input to the filter mechanism. This provides the ability for our assurance pattern to identify any design flaws. The solution node '*chkFilterBypass*' of our assurance pattern as shown in Figure 21 captures this information as evidence for supporting the sub-claim.

To ensure that the design is appropriately zero-trust, the assurance pattern fragments are not only provided as part of individual zero-trust mechanisms, these patterns are also provided in an assurance pattern library. Engineers can easily utilize them while designing system models depending on the zero-trust system requirements. Please note that if there are multiple zero-trust components of the same type present in a model, the corresponding assurance pattern will be instantiated for each one. For instance, if multiple attestation components are present in a system model, to ensure that the design is correct for each attestation mechanism, the above mentioned sub-goals for the attestation mechanism are iterated over each attestation mechanism in the model.

Please note that all of our design patterns are represented in AADL, however these design patterns can be represented in any architecture language (e.g., SysML) provided they follow the correct representation.

IV. Zero-Trust Assurance of a UAV Model

In this section, we will consider an example UAV surveillance application to demonstrate our zero-trust design and assurance patterns. The UAV architecture is represented in Figure 22. It consists of a flight controller (as shown in green), mission computer (light orange), network buses for data transfer (white and light purple), and a camera (orange). The mission computer further consists of the hardware devices such as radio, WiFi, etc., (as shown in gray). In addition, it also hosts the software (light green) responsible for the functionality and behavior of the UAV.

At the system level, we add encryption buses to the UAV model as represented by purple color in Figure 22. The light green element represents the higher-level representation of the UAV mission computer software. Zero-trust components are added to the software (not visible at this abstraction level). Figure 23 shows the mission computer software architecture model with the addition of the zero-trust elements (shown in light purple). Here, an attestation mechanism is added to the UART driver to attest the connected device, i.e., the flight control computer in this case. Further, a PEP is attached to the UART driver to enable a communication bridge between the

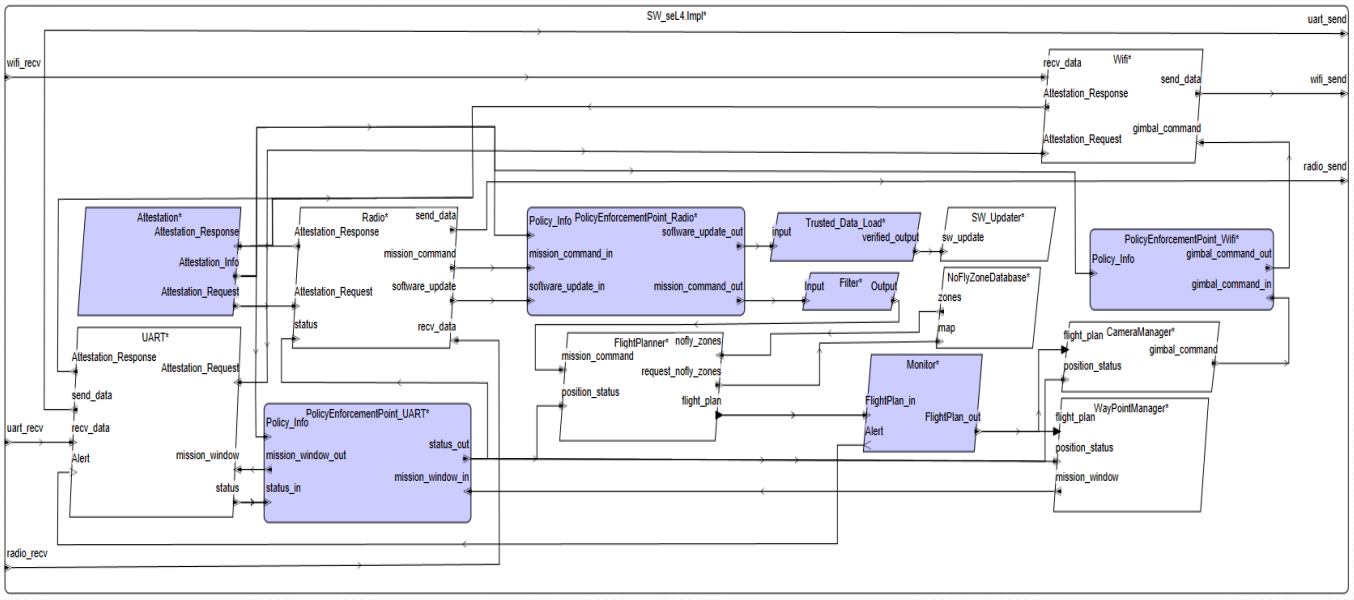

Fig. 23. Software module of the UAV model in AADL (light purple color represents the zero-trust mechanisms).

```
70      goal overallSystemCompliance(s : system) <=
71          ** "The system is ZTA compliant" **
72          strategy_OverallSystemCompliance(s)
73
74      Strategy strategy_OverallSystemCompliance(s : system) <=
75          ** "System security is addressed using various ZTA mechanisms and features" **
76          separationKernelCompliance(s)
77          and attestationCompliance(s)
78          and encryptionCompliance(s)
79          and trustedDataLoadCompliance(s)
80          and runtimeMonitorCompliance(s)
81          and filterCompliance(s)
82          and policyEnforcementPointCompliance(s)
```

Fig. 24. Top-level assurance pattern in Resolute.

UART driver and other components such as the flight planner only if the trust is established. Similarly, other mechanisms are added at specific places in the model and are responsible for performing desired functions as described in Section III.

Next, we construct our assurance case using the assurance pattern fragments introduced in Section III. Figure 24 shows the top-level goal that utilizes the individual zero-trust patterns defined in Section III. Figure 25 represents the attestation mechanism's assurance pattern fragment and is defined in OSATE using Resolute. As seen in Figure 25, the pattern fragment follows the same definition as discussed in Section III. Here we have shown the textual representation of the assurance fragment to demonstrate the collection of evidence from the system architecture, as seen on lines 122-123. The functions defined in these lines are not shown here but they are responsible for collecting evidence either directly from the system architecture or via other external tools. These patterns follow the GSN standard and can be easily exported to a graphical tool as well.

In the subsections that follow, we describe the importance of our approach from two perspectives: 1) we demonstrate how the same zero-trust assurance patterns defined in Section III and included in our assurance pattern library can be utilized to verify different zero-trust system designs as per their individual security requirements, and 2) we also showcase the ability of our tool to capture any zero-trust transformation gaps resulting in violations of model compliance with respect to the zero-trust system design.

A. Zero-Trust Design Verification

In this scenario, the zero-trust system requirements consist of all the zero-trust mechanisms described in Section III. The top-level goal for assurance of each mechanism is also shown in Figure 24 (lines 76-82). Using the zero-trust requirements as the base, we have added the required zero-trust mechanisms to the UAV model as shown in Figures 22 and 23. Next,

```
goal attestationCompliance(s : system) <=
    ** "Attestation mechanism compliance is achieved" **
    strategy_AttestationCompliance(s)

Strategy strategy_AttestationCompliance(s : system) <=
    ** "System security using attestation mechanism is addressed" **
    let commDrivers : {component} = {c for (c : component) | has_property(c, ZTA_Properties::Component_Type)
        and property(c, ZTA_Properties::Component_Type) = "COMM_DRIVER"
    };
    let commDrivers_list : {component} = {c for (d : commDrivers) (c : subcomponents(d))};
    let commDrivers_of_interest : {component} = {d for (d : commDrivers) | not (member(d, commDrivers_list))};
    let d1 : bool = debug("DRIVERS= ", commDrivers_of_interest);
    forall (driver : commDrivers_of_interest) . (checkAttestationCompliance(driver))

goal checkAttestationCompliance(driver : component) <=
    ** "Attestation criteria met" **
    let peps : {component} = {c for (c : component) | has_property(c, ZTA_Properties::Component_Type)
        and property(c, ZTA_Properties::Component_Type) = "POLICY_ENFORCEMENT_POINT"
    };
    let pep : {component} = {c for (c : peps) | exists(conn : connection) . (is_source_component(conn, driver)
        and is_destination_component(conn, c)
    ) or (is_source_component(conn, c)
        and is_destination_component(conn, driver))};
    let attestations : {component} = {c for (c : component) | has_property(c, ZTA_Properties::Component_Type)
        and property(c, ZTA_Properties::Component_Type) = "ATTESTATION"
    };
    let attestation : {component} = {c for (c : attestations) | exists(conn : connection) . (is_source_component(conn, driver)
        and is_destination_component(conn, c)
    )};
    length(pep) >= 1 and length(attestation) = 1 and existsAttestation(driver, pep, head(as_list(attestation)))
        and attestationNotBypassed(driver, pep) and attestationImplementationCorrectness(head(as_list(attestation)))
```

Fig. 25. Attestation requirement assurance pattern fragment in Resolute.

we utilize the assurance pattern fragments for individual zero trust mechanisms and knit them together to support the top-level goal for achieving overall zero-trust system compliance. Finally, the assurance pattern is instantiated with the UAV model and an assurance argument is generated using Resolute. We further demonstrate two scenarios; a passing and a failing assurance argument. The passing assurance argument confirms the necessary evidence exists to support the top-level zero-trust compliance goal, whereas a failing assurance argument successfully demonstrates the ability of our tool to identify design violations due to lack of evidence. We further discuss each of these scenarios in a bit more detail:

1) Passing Assurance Argument: Figure 26 shows the passing assurance argument. This argument is generated by Resolute, represented in a tree-like structure. Each node represents a goal that is satisfied using some leaf-level evidence. The green check marks imply that evidence was captured to support the goal associated with it. In our assurance argument, the supporting evidence is captured using the architecture model of the UAV. Resolute provides the ability to capture this type of evidence by walking over the AADL model. As shown in Figure 26, all the required evidence was captured and the top-level assurance goal was successfully supported, thereby generating a passing argument. This argument supports the claim that the UAV model is zero-trust compliant after its transformation, meaning the model satisfies its zero-trust requirements.

To further demonstrate how these patterns are utilized together, we have also generated the graphical representation of the assurance argument represented in Figure 26. The graphical representation is shown in Figure 28. Because the full assurance argument is quite large, it is not legible in this format, but the figure is labeled with individual assurance fragments to describe each argument branch for ease of understanding. Depending upon the requirements of the zero-trust model, respective assurance pattern fragments such as attestation, encryption, etc., as discussed in Section III are knit together to support the top-level goal. Each of these assurance arguments are instantiated with a specific system context to collect the required evidence. Furthermore, the collected evidence represented as green circles for individual pattern fragments are used to support the overall goal. These fragments are reusable and can be utilized in various configurations depending on the system's zero-trust requirements to satisfy the top-level claim.

2) Failing Assurance Argument: In this scenario, we have modified the UAV model to violate the design assurance for the filter mechanism. Specifically, we added a connection that bypasses the filter input. This is one of the design assurance sub-goals discussed in Section III. Due to this change the UAV model no longer satisfies the zero-trust architecture requirements. As a result, when we utilize our assurance pattern from the pattern library and generate an assurance argument to evaluate our design, Resolute is able to capture this assurance violation and notify the user. Figure 27 shows the resulting assurance argument. As seen in the assurance tree represented by Resolute, the filter bypass property is violated (shown by red exclamation marks in the figure). This propagates up the assurance argument tree and results in an invalid assurance argument overall. Similarly, any other design assurance violations can be captured by Resolute and appropriate measures can be taken to ensure system requirements are satisfied.

Fig. 26. Passing assurance argument demonstrating evidence for zero-trust requirement satisfaction in the UAV model.

Fig. 27. Failing assurance argument demonstrating missing evidence for zero-trust requirements in the UAV model.

B. Zero-Trust Design Verification With No Monitoring Requirement

In this scenario, the zero-trust system requirements consist of all the zero-trust mechanisms described in Section III other than the run-time monitor. Therefore, we have added all the zero-trust mechanisms to the UAV model and eliminated the monitoring element due to the new requirement. The top-level assurance goal for this scenario will be similar to the one described in Figure 24, but line 80 will not be part of this revised goal due to the elimination of the monitor element from the design requirement. This means that we will not be evaluating the assurance fragment associated with the monitor element.

The top-level goal references the assurance pattern fragments for the individual zero-trust mechanisms and knits them together to support the top-level argument pattern that focuses on achieving an overall zero-trust compliant system. The assurance pattern was instantiated using the UAV model as the context and an assurance argument was generated. Since there is no requirement violation, our passing assurance case looks similar to Figure 26. The only difference is that it will not contain an argument that the system is properly monitored. To demonstrate this appropriately, we generated the graphical representation of the assurance argument. Figure 29 shows the graphical representation of the passing assurance argument for the resulting zero-trust system design as per the new zero-trust security requirements. As seen from Figure 29, there is no assurance fragment for the monitor element. However, the system is successfully verified as all the other security requirements are met. Any design violations in any of the other zero-trust requirements will be identified in a similar way as described in the above failing assurance scenario.

The above scenarios clearly demonstrate the use of our zero-trust assurance pattern library in composing and verifying a zero-trust compliant system design based on individual requirements and capturing any design flaws such that they can be handled appropriately early in the design phase.

V. Conclusions and Future Work

Security is becoming a challenge in aerospace and other mission-critical domains. In order to address this challenge,

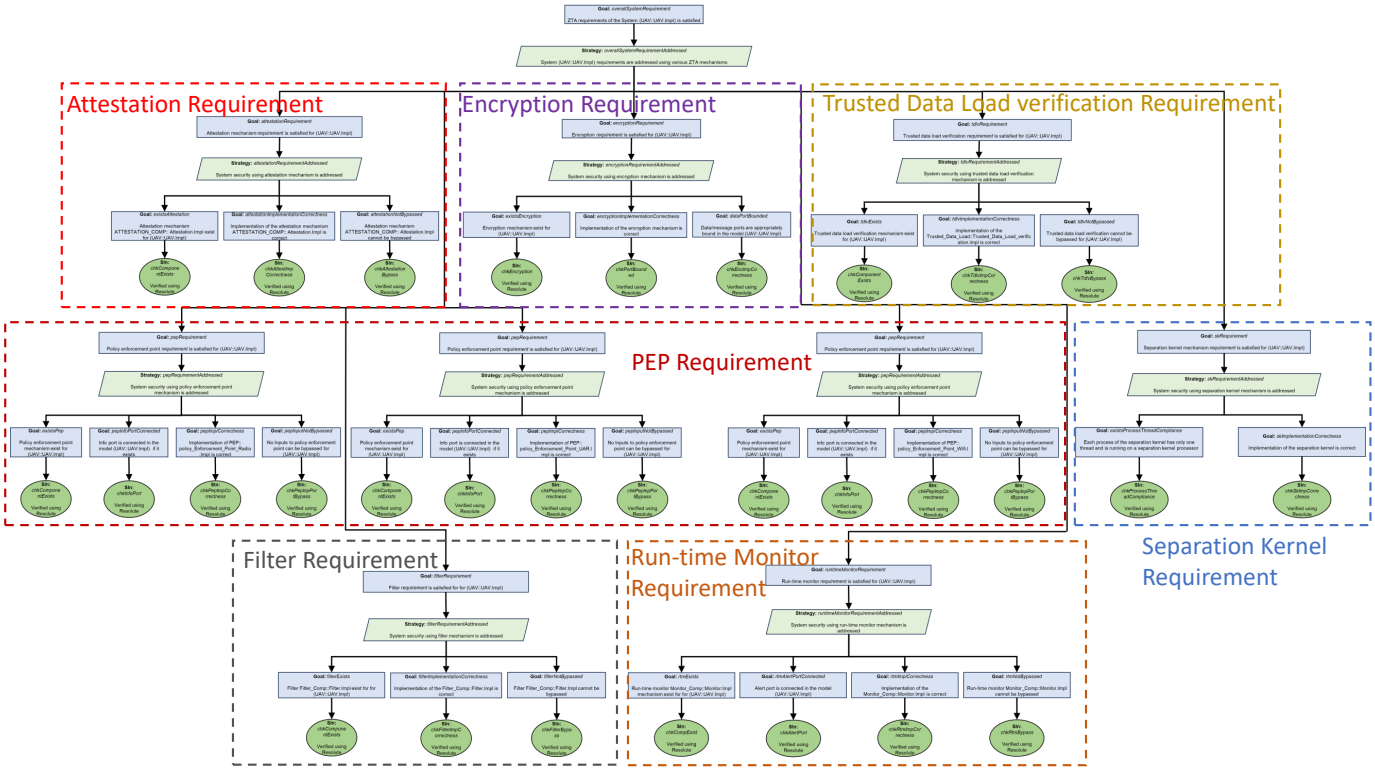

Fig. 28. Graphical representation of a passing assurance argument for zero-trust compliance in the UAV model.

new initiatives such as zero-trust are being researched. Zero-trust has proved successful in the Information Technology domain; however, lack of zero-trust mechanisms geared specifically towards CPS have delayed adoption of zero-trust principles in these domains. Moreover, modeling and analysis mechanisms in addition to tools that can support assurance for the resulting system design are presently non-existent for zero-trust. We have identified an initial set of zero-trust design and assurance patterns that can be utilized to build systems based on the concepts that reflect the core tenets of zero-trust. Additionally, to ensure a zero-trust enabled system design, it is useful to link a system design model with an assurance analysis framework. We have developed our zero-trust assurance patterns in the Resolute language and leverage the BriefCASE framework [33], which utilizes the system architecture modeling environment to evaluate assurance. The resulting analysis artifacts can report any design violations to engineers when needed.

To make the overall process easier, we have provided these design patterns and their assurance pattern fragments as individual libraries that can be easily utilized by engineers. Engineers can employ one or more patterns and evaluate the system-level assurance based on the overall zero-trust requirements of a given system. Finally, we have demonstrated the feasibility of our approach using a UAV surveillance application. We discussed how our assurance patterns can be utilized in combination to provide assurance based on different zero-trust requirements. In addition, we also demonstrated the usefulness of this approach in empowering engineers to identify system design flaws and correct them to save significant amount of development time, effort, and cost. We have also pointed out the use of our approach to support the certification process, if necessary.

As part of future work, we will further develop our initial zero-trust assurance patterns and provide automation for the overall process. To elaborate further, our vision is to create a tool which takes zero-trust system requirements as input from the user and automatically inserts appropriate zero-trust mechanisms into an untrusted system model, thus generating a zero-trust enabled system. Further, we would like to automatically invoke specific zero-trust assurance pattern fragments to validate design requirements. To achieve this, we will develop injection rules that will be utilized by our tool to auto-insert specific zero-trust mechanisms into the model. In addition, these rules will leverage the tags or annotations that will be required as part of the entire automation framework. We believe that such a tool will be very useful in designing more secure zero-trust enabled systems while providing the necessary design assurance.

References

[1] Stephenson Harwood. *Aviation is facing a Rising Wave of Cyber-Attacks in the Wake of COVID*. Accessed: 01-30-2024. 2022. URL: https://www.shlegal.com/insights/aviation-is-facing-a-rising-wave-of-cyber-attacks-in-the-wake-of-covid.

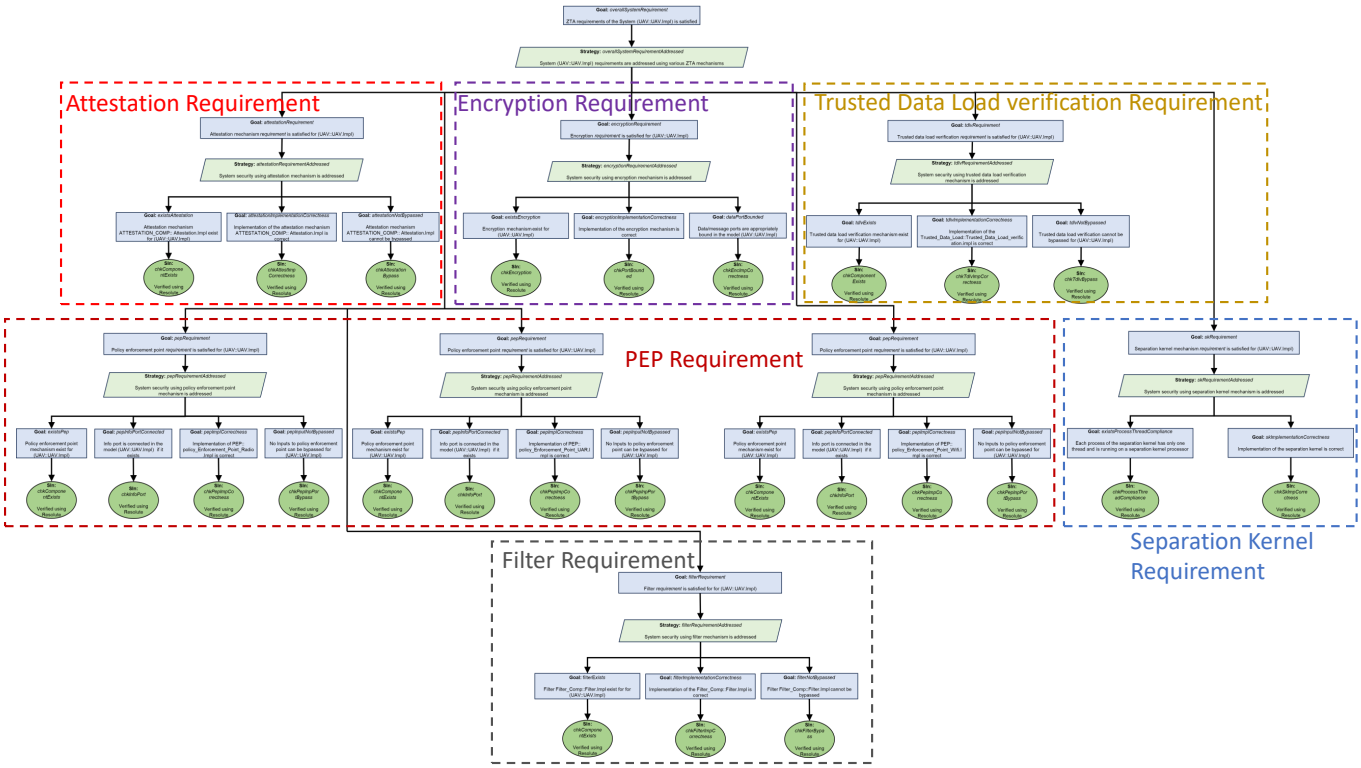

Fig. 29. Graphical representation of a passing assurance argument for zero-trust requirement verification of a UAV model with no monitoring element.

[2] KonBriefing. *Cyber attacks on the aviation industry in 2022, Statistics: Ransomware, data breaches, DDoS attacks*. Accessed: 01-30-2024. 2023. URL: https://konbriefing.com/en-topics/cyber-attacks-2022-ind-aviation.html.

[3] TechForce. *Case Study: Cyberattacks in the Aviation Industry*. Accessed: 01-30-2024. URL: https://techforce.co.uk/blog/2023/case-study-cyberattacks-in-the-aviation-industry---risks-and-remedies.

[4] Yuzuka Akasaka. *Top Cyber Threats Faced by the Aviation Industry*. Accessed: 01-30-2024. 2023. URL: https://securityboulevard.com/2023/06/top-cyber-threats-faced-by-the-aviation-industry/.

[5] Adam Janofsky. *Cyber incident at Boeing subsidiary causes flight planning disruptions*. Accessed: 01-30-2024. 2022. URL: https://therecord.media/cyber-incident-at-boeing-subsidiary-causes-flight-planning-disruptions.

[6] Angelo Mathais. *SpiceJet's woes continue as ransomware attack delays flights*. Accessed: 01-30-2024. 2022. URL: https://theloadstar.com/spicejets-woes-continue-as-ransomware-attack-delays-flights/.

[7] Howard Soloman. *Canadian military provider suffered ransom attack, says news report*. Accessed: 01-30-2024. 2022. URL: https://www.itworldcanada.com/article/canadian-military-provider-suffered-ransom-attack-says-news-report/487654.

[8] Saqib Hasan, Isaac Amundson, and David Hardin. *Zero Trust Architecture Patterns for Cyber-Physical Systems*. Tech. rep. SAE Technical Paper, 2023.

[9] Richard Hawkins et al. "A new approach to creating clear safety arguments". In: *Advances in Systems Safety: Proceedings of the Nineteenth Safety-Critical Systems Symposium, Southampton, UK, 8-10th February 2011*. Springer. 2011, pp. 3–23.

[10] Yaping Luo, Zhuoao Li, and Mark Van Den Brand. "A categorization of GSN-based safety cases and patterns". In: *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. IEEE. 2016, pp. 509–516.

[11] Mario Gleirscher and Carmen Carlan. "Arguing from hazard analysis in safety cases: a modular argument pattern". In: *2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*. IEEE. 2017, pp. 53–60.

[12] Richard Hawkins et al. "Guidance on the assurance of machine learning in autonomous systems (AMLAS)". In: *arXiv preprint arXiv:2102.01564* (2021).

[13] Charles B Weinstock, Howard F Lipson, and John Goodenough. *Arguing security-creating security assurance cases*. Tech. rep. Carnegie Mellon University, 2007.

[14] Richard Hawkins, Tim Kelly, and Ibrahim Habli. "Developing Assurance Cases for D-MILS Systems." In: *MILS@ HiPEAC*. 2015.

[15] Irfan Šljivo et al. “Guiding assurance of architectural design patterns for critical applications”. In: *Journal of Systems Architecture* 110 (2020), p. 101765.

[16] Scott Rose et al. “NIST special publication 800–207 zero trust architecture”. In: *National Institute of Standards and Technology, US Department of Commerce* (2020), pp. 800–207.

[17] “Department of Defense (DOD) Zero Trust Reference Architecture”. In: (). URL: https://dodcio.defense.gov/Portals/0/Documents/Library/(U)ZT_RA_v1.1(U)_Mar21.pdf.

[18] David A Wheeler, EK Fong, and Institute for Defense Analyses Alexandria. *A Sample Security Assurance Case Pattern*. Tech. rep. 2018.

[19] Clément Duffau, Thomas Polacsek, and Mireille Blay-Fornarino. “Support of justification elicitation: Two industrial reports”. In: *Advanced Information Systems Engineering: 30th International Conference, CAiSE 2018, Tallinn, Estonia, June 11-15, 2018, Proceedings 30*. Springer. 2018, pp. 71–86.

[20] “Goal Structuring Notation Community Standard Version 3”. In: (). URL: https://scsc.uk/r141C:1?t=1.

[21] Peter H Feiler and David P Gluch. *Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language*. Addison-Wesley, 2012.

[22] SEI AADL Team et al. *OSATE: Plug-ins for front-end processing of AADL models*. 2008.

[23] Isaac Amundson and Darren Cofer. “Resolute assurance arguments for cyber assured systems engineering”. In: *Proceedings of the Workshop on Design Automation for CPS and IoT*. 2021, pp. 7–12.

[24] Robert Sheldon, Peter Loshin, and Michael Cobb. *What is data security? The ultimate guide*.

[25] SP NIST. “800-38A:“Recommendation for Block Cipher Modes of Operation”. In: *Methods and Techniques”, NIST* (2001).

[26] Elaine Barker. “NIST Special Publication 800–175B Revision 1 Guideline for Using Cryptographic Standards in the Federal Government: Crypto-graphic Mechanisms”. In: *Computer Science* (2018), pp. 1–83.

[27] Erdem. *What’s the Difference between Secure Boot and Measured Boot?* 2015.

[28] Jämes Ménétrey et al. “Attestation mechanisms for trusted execution environments demystified”. In: *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer. 2022, pp. 95–113.

[29] VA Stafford. “Zero trust architecture”. In: *NIST special publication* 800 (2020), p. 207.

[30] Malte Mauritz, Falk Howar, and Andreas Rausch. “Assuring the safety of advanced driver assistance systems through a combination of simulation and runtime monitoring”. In: *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications: 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II 7*. Springer. 2016, pp. 672–687.

[31] David Cooper et al. “Security considerations for code signing”. In: *NIST Cybersecurity White Paper* (2018).

[32] Gerwin Klein et al. “seL4: Formal verification of an OS kernel”. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 2009, pp. 207–220.

[33] Darren Cofer et al. “Cyberassured systems engineering at scale”. In: *IEEE Security & Privacy* 20.3 (2022), pp. 52–64.