Zero Trust Architecture Patterns for Cyber-Physical Systems

Saqib Hasan, Isaac Amundson, David Hardin
Collins Aerospace, Applied Research and Technology

Abstract

Zero trust (ZT) is an emerging initiative that focuses on securely providing access to resources based on defined policies. The core tenet of ZT is "never trust, always verify", meaning that even within trusted zones of operation, resource access must be explicitly granted. ZT has proven effective in improving the security posture in domains such as information technology infrastructure; however, additional research and development is needed to define and apply zero trust principles to cyber-physical system domains. To work toward this objective, we have identified an initial set of ZT architectural patterns targeted specifically at cyber-physical systems. We created ZT architecture patterns in the Architecture Analysis and Design Language (AADL), a modeling language that enables engineers to describe the key elements of embedded system architectures using a well-defined semantics. The patterns are implemented as a library of ZT components that can be made available to system engineers. Utilizing AADL capabilities, engineers can model a system in AADL and apply one or more of these ZT patterns to improve the system security posture based on specific system requirements. To demonstrate our approach, we apply the ZT patterns to an unmanned aerial vehicle surveillance application. The resulting design provides inherent protection from a variety of attacks affecting system confidentiality, integrity, and availability.

Introduction

Cyber-physical systems (CPS) are complex systems comprised of various hardware and software components. These components communicate to provide the overall behavior that satisfies system requirements. However, the scale and complexity of these systems can give rise to various cyber vulnerabilities. Adversaries can exploit such vulnerabilities and cause catastrophic consequences resulting in system damage, loss of infrastructure, failure of critical missions, financial impact, etc. For instance, in 2015, security researchers demonstrated gaining control of a Jeep remotely and shutting it down while being driven on the highway. Another recent incident suggests a group of attackers were able to steal several vehicles by gaining access to the keyless entry and start system [1]. Aircraft, being a class of CPS, are not immune to such cyber vulnerabilities; a recent report suggests that the aviation industry is facing a growing wave of cyber-attacks [2]. Hence, it becomes paramount to develop tools and technologies that improve CPS security.

Zero Trust (ZT) is an emerging initiative that focuses on securely providing access to resources based on defined policies. The core tenet of ZT is "never trust, always verify", meaning that even within trusted zones of operation, resource access must be explicitly granted. ZT has proven effective in improving the security posture in domains such as information technology (IT) infrastructure, however, additional research and development is needed to define and apply zero trust principles to cyber-physical system domains such as aerospace.

A reference architecture for zero trust is provided in [3]. The work presented in [4] also provides a detailed description of zero trust architecture. It describes the core ZT tenets, variations in ZT architectures, and their applicability from the perspective of IT infrastructure. Authors in [5] discusses the importance of cyber security in various domains and the need for better technologies to address it. As per their research, zero trust is a key emerging technology that changes the cybersecurity approach; however, more work is needed to apply ZT to specific domains. Work presented in [6] provides a new zero trust model for embedded systems. In this work, the author discusses the importance of a separation environment using ZT principles to provide improved security. Additional research presented in [7] provides a security and safety risk analysis method based on zero trust. This work discusses the importance of safety and security design weakness identification at an early stage. This research provides a framework called the Multidisciplinary Early Design Risk Assessment Framework (MEDRAF) for performing early risk assessment including both safety and security aspects.

Based on our research, no tools or methods currently provide a means to model system architectures using ZT patterns. Specifically, we lack consensus on a definition of ZT that is applicable to CPS. In addition, once defined, how does the resulting approach become useful for the CPS domain? To work toward these objectives, the paper provides the following contributions:

1. We have identified an initial set of ZT mechanisms targeted specifically for cyber-physical systems.
2. We created corresponding ZT architecture patterns in the Architecture Analysis and Design Language (AADL), a

Pre-print for the following paper published by the SAE:

Hasan, S., Amundson, I., and Hardin, D., "Zero Trust Architecture Patterns for Cyber-Physical Systems," SAE Technical Paper 2023-01-1001, 2023, doi:10.4271/2023-01-1001.

modeling language that enables engineers to describe the key elements of embedded system architectures using a well-defined semantics [8].

3. The patterns can be implemented as a library of ZT components that can be made available to system engineers. Utilizing AADL capabilities, engineers can model systems in AADL and apply one or more of these ZT patterns during the system design phase to improve the overall system security posture based on specific system requirements.
4. To demonstrate our approach, we apply the ZT patterns to an unmanned aerial vehicle (UAV) surveillance application and discuss how the inclusion of our ZT mechanisms can prevent cyber-attacks from affecting the security of the overall system.

We have utilized AADL as the design language to achieve our objective of implementing identified ZT patterns. AADL is an architecture language that enables modeling of real-time distributed embedded systems while capturing important design concepts [9]. Therefore, it is well suited to model avionics system models, for example. AADL provides the ability to capture both hardware and software architecture details hierarchically. Hardware components include memory, buses, processors, and devices whereas software components include processes, threads, subprograms, and data. In addition, the language provides the ability to define data flows, connections, interfaces, and properties. AADL provides a high degree of flexibility that enables incremental model-based development such that the architectures and the components can be refined over time. The Open Source AADL Tool Environment (OSATE) [10] is the reference framework for AADL, which we use to build our UAV surveillance system and represent our library of ZT patterns.

Although our ZT pattern library can be used as a collection of reference models for the manual composition of cyber-resilient architectures, we envision the library being integrated with a cybersecurity-focused development toolchain such as BriefCASE [11]. BriefCASE includes tools for analyzing an architecture model for cyber vulnerabilities, mitigating the vulnerabilities by applying automated model transformations, synthesizing provably correct code, and automatically generating a cybersecurity assurance case using elements found within the model as supporting evidence. With BriefCASE, if a new requirement specifies the need for an additional ZT protection, the tool can automatically modify the architecture model with the corresponding ZT pattern. BriefCASE will then produce design assurance that each ZT mechanism in the model has been inserted correctly (e.g., the ZT component cannot be bypassed) using the Resolute assurance tool [13]. This novel approach to designing inherently cyber-resilient CPS was developed on the DARPA CASE program with the goal of equipping systems engineers with new tools and methods to reason about cybersecurity concerns in high-assurance systems.

The paper is organized as follows. First, we provide a brief background on ZT. Next, we discuss the identified ZT mechanisms in detail, and propose a corresponding AADL pattern for each. We next provide details of our approach using an experimental UAV model. Finally, we provide a conclusion and discuss future work.

Zero Trust Background

Zero Trust is a “cybersecurity strategy developing an architecture that requires authentication or verification before granting access to sensitive data or protected resources at a financial cost by reducing data loss and preventing data breaches” [3]. Zero trust relies on several core tenets such as: assume a hostile environment (all devices and networks are considered untrusted); presume breach (presence of an adversary is considered within the operational environment and proactive scrutiny is performed for access and authorization); never trust, always verify (access is only provided to a resource after explicitly authenticating the device); scrutinize explicitly (change access policies of resources dynamically based on confidence levels and actions); and apply unified analytics (use analytics to improve and support access policies). These are discussed in [3], an emerging initiative that is being explored by a collaborative effort between DOD CIO, National Security Agency (NSA), DISA, and US Cyber Command (USCYBERCOM).

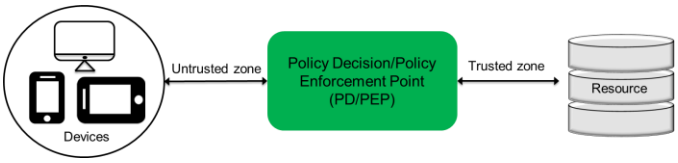

Figure 1. Zero trust architecture concept.

Figure 1 represents the basic structure of a zero-trust architecture. Here, an untrusted entity requires access to a resource. The policy decision and enforcement point (PD/PEP) receives the access request and provides a decision whether to allow or deny access to the requested resource. This decision is based on authentication and authorization policies. In addition, the PD/PEP also receives information from the environment and other systems that can be utilized in augmenting the decision process. For instance, in order to identify if the requester is a legitimate entity, the PD/PEP can collect device information and match it with the existing device identities in its database.

Identified ZT Mechanisms

Zero trust mechanisms provide a means of improving the security of the overall system. When included in an architecture model they can result in a more cyber-resilient system. In addition, their usage in the model depends on several factors such as level of security needed to be achieved, the number of critical resources to be protected, available budget to deploy in order to protect resources from security threats, infrastructure maintenance cost, etc. In this section, we describe the ZT mechanisms that support our approach, as well as document their representation in AADL.

Secure Data Load

Secure data load ensures that data load authenticity and integrity is maintained. This consists of two components, *encryption* and *decryption*. Both components maintain data authenticity and integrity during individual phases of the data load process. We describe each of these components in detail as follows:

Encryption

The purpose of an encryption mechanism is to ensure data authenticity and integrity by encrypting data at the point of origination. Figure 2 shows the encryption mechanism, which consists of an `Encryption_Manager` and an `Encryption_Policy` component. Rules for encryption are defined as a security policy in the `Encryption_Policy` component. Based on these rules, when data is received at its input

port, the `Encryption_Policy` triggers the `Encryption_Manager` to perform encryption. The `Encryption_Manager` utilizes the `Cryptographic_Keys` and `Cryptographic_Algorithm` to encrypt the data and sends it back to the `Encryption_Policy` component, which then passes the encrypted data as output.

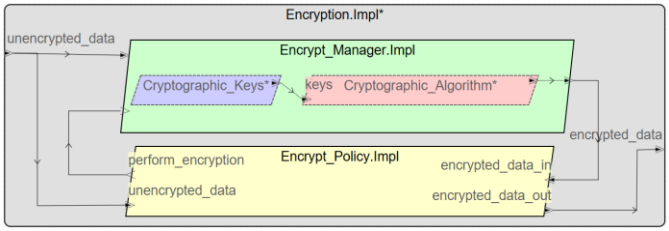

Figure 2. Encryption mechanism architecture.

Figure 3 shows the AADL textual representation of the encryption mechanism. In AADL, there is no agreed-upon way to model the communication protocol details. AADL provides a *bus* element for modeling physical buses. These buses are utilized to represent the connections and information flow between different components in the system model. However, implementation details (e.g., whether the bus provides secure data transfer) cannot natively be specified on buses without applying a custom property association. Moreover, elements that could represent details (such as layers of the communication stack) are not defined in a standard way. From the communication protocols standpoint, we know that the application layer of the TCP/IP protocol is responsible for performing the encryption/decryption mechanisms on the messages that are being transferred. We have modeled such mechanism using virtual buses in AADL (lines 36-39). First, we bound these buses to individual physical buses (lines 57-59) to represent the application layer in the communication protocol that is responsible for data security. Next, the connections that are supposed to carry encrypted messages within the system are bounded to these virtual buses, meaning only the connections which are bound to the virtual buses can carry encrypted information in order to support the functionality of application layer protocols such as SSL/TLS (lines 60-62). The connections which are directly bound to physical buses, or any other buses cannot be considered secure. Although, it is possible to model a complete network stack in AADL we have decided to keep our implementation simple and used AADL property associations to capture the necessary aspects of encryption.

```
28⊖   system implementation UAV.Impl
29        subcomponents
30            MCMP: system MC::MissionComputer.Impl;
31            FCTL: system FC::FlightController.Impl;
32            SBUS: bus UAS_Buses::Serial_Bus.Impl;
33            CAM: device Camera.Impl;
34            WBUS: bus UAS_Buses::Wifi_Bus.Impl;
35            Radio_Bus: bus UAS_Buses::RF_Bus.Impl;
36            NETWORK_BUS_Radio: virtual bus UAS_Buses::SSL_TLS.Impl;
37            NETWORK_BUS_Wifi: virtual bus UAS_Buses::SSL_TLS.Impl;
38            NETWORK_BUS_UART: virtual bus UAS_Buses::SSL_TLS.Impl;
39        connections
40            bac1: bus access SBUS <-> MCMP.UARTA;
41            bac2: bus access SBUS <-> FCTL.UARTA;
42            bac3: bus access RFA -> MCMP.RFA;
43            bac4: bus access WBUS <-> MCMP.WIFIA;
44            bac5: bus access WBUS <-> CAM.WIFIA;
45            bac6: bus access Radio_Bus <-> MCMP.RFA;
46            bac8: virtual bus access MCMP.SSL_TLS <-> NETWORK_BUS_Radio;
47            bac9: virtual bus access MCMP.SSL_TLS <-> NETWORK_BUS_Wifi;
48            bac10: virtual bus access MCMP.SSL_TLS <-> NETWORK_BUS_UART;
49            c1: port radio_recv -> MCMP.radio_recv;
50            c2: port MCMP.radio_send -> radio_send;
51            c3: port MCMP.uart_send -> FCTL.uart_recv;
52            c4: port FCTL.uart_send -> MCMP.uart_recv;
53            c5: port MCMP.wifi_send -> CAM.wifi_recv;
54            c6: port CAM.wifi_send -> MCMP.wifi_recv;
55        properties
56            Actual_Connection_Binding => (reference (Radio_Bus)) applies to NETWORK_BUS_Radio;
57            Actual_Connection_Binding => (reference (SBUS)) applies to NETWORK_BUS_UART;
58            Actual_Connection_Binding => (reference (WBUS)) applies to NETWORK_BUS_Wifi;
59            Actual_Connection_Binding => (reference (NETWORK_BUS_Radio)) applies to c1, c2;
60            Actual_Connection_Binding => (reference (NETWORK_BUS_UART)) applies to c3, c4;
61            Actual_Connection_Binding => (reference (NETWORK_BUS_Wifi)) applies to c5, c6;
62
63⊖           annex resolute {**
64                argue overallSystemCompliance(this)
65            **};
66    end UAV.Impl;
```

Figure 3. AADL representation of the encryption mechanism.

Decryption

Figure 4 represents the decryption mechanism architecture. The purpose of decryption is to securely decrypt data at its destination into its original form. The architecture of the decryption mechanism is similar to encryption, however, instead of `Encryption_Policy` and `Encryption_Manager` components, it utilizes a `Decryption_Policy` and a `Decryption_Manager` to perform the necessary operations. Once, encrypted data appears at the input port of the decryption mechanism, it decrypts the encrypted data using the `Cryptographic_Keys` and `Cryptographic_Algorithm` contained in the `Decryption_Manager` component. The decrypted data is then passed as an output via the `Decryption_Policy` component of the decryption mechanism.

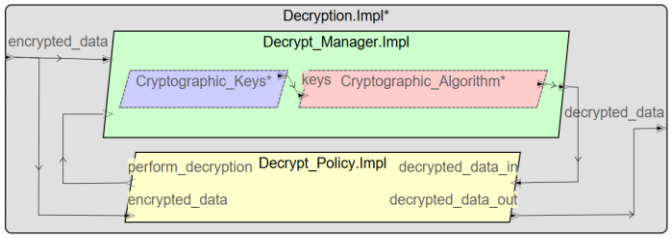

Figure 4. Decryption mechanism architecture.

The AADL textual representation of the decryption mechanism is similar to the encryption mechanism and therefore is not shown here.

Attestation

An attestation component ensures software authenticity and integrity. Figure 5 shows the attestation mechanism. It utilizes two sub-components, `Secure_Boot` and `Measured_Boot`, to perform attestation. The rule for a successful attestation is defined by the `Attestation_Policy`. `root_of_trust` is used as an input for both secure and measured boot. These processes generate individual data structures such as `validation` and `quote`. The `validation` message contains information regarding software integrity. This could represent binary information that corresponds to software authenticity and integrity. `quote` is a data structure that contains detailed information about this software. For instance, it could contain information such as software version. Both are further utilized by the `Attestation_Policy` to make an attestation decision.

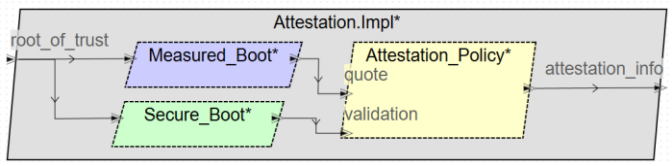

Figure 5. Attestation architecture pattern.

Figure 6 shows the textual AADL representation of the attestation mechanism. In AADL each mechanism has two parts, namely a component type and its implementation. The component type specifies the component interface, properties, flows, inheritance, etc. The implementation describes specific subcomponents and how they are connected. A component can be implemented in different ways;

thus, an AADL model could contain multiple component implementations for the same component type, each implementation having different subcomponents, connections, properties, etc. First, we define the component type (lines 33-39) of the attestation mechanism. The mechanism consists of various features (lines 35-36) which represent the input/output ports of the component. Next, we define the implementation of the attestation mechanism. This describes the details of its sub-components and their connections (lines 42-51). Each of the sub-components are defined as another AADL element referred to as a thread. In order to identify the attestation component in the model we have set a property as defined in line 38. This property will be useful for specific analyses performed on system models containing attestation components.

Figure 7 shows the AADL representation of the attestation sub-components. Sub-components such as `Secure_Boot`, `Measured_Boot`, and `Attestation_Policy` are defined as threads in our implementation with their own component types and implementations. All these components consist of features that align with the design discussed above. Note that these components could also be implemented using different AADL components (such as system or abstract components) depending upon user needs and requirements and are not limited to the specific AADL types shown in our implementation.

```
33  process Attestation
34      features
35          root_of_trust: in event data port;
36          attestation_info: out event data port;
37      properties
38          ZTA_Properties::Component_Type => ATTESTATION;
39  end Attestation;
40
41  process implementation Attestation.Impl
42      subcomponents
43          Secure_Boot: thread Sec_Boot.Impl;
44          Measured_Boot: thread Mea_Boot.Impl;
45          Attestation_Policy: thread Att_Policy.Impl;
46      connections
47          c1: port root_of_trust -> Secure_Boot.root_of_trust;
48          c2: port root_of_trust -> Measured_Boot.root_of_trust;
49          c3: port Secure_Boot.validation -> Attestation_Policy.validation;
50          c4: port Measured_Boot.quote -> Attestation_Policy.quote;
51          c5: port Attestation_Policy.attestation_info -> attestation_info;
52  end Attestation.Impl;
```

Figure 6. AADL textual representation for attestation.

```
5   thread Sec_Boot
6       features
7           root_of_trust: in event data port;
8           validation: out event data port;
9   end Sec_Boot;
10
11  thread implementation Sec_Boot.Impl
12  end Sec_Boot.Impl;
13
14  thread Mea_Boot
15      features
16          root_of_trust: in event data port;
17          quote: out event data port;
18  end Mea_Boot;
19
20  thread implementation Mea_Boot.Impl
21  end Mea_Boot.Impl;
22
23  thread Att_Policy
24      features
25          validation: in event data port;
26          quote: in event data port;
27          attestation_info: out event data port;
28  end Att_Policy;
29
30  thread implementation Att_Policy.Impl
31  end Att_Policy.Impl;
```

Figure 7. AADL textual representation of the attestation sub-components.

Policy Enforcement Point (PEP)

Figure 8 represents the architecture diagram for the policy enforcement point mechanism. The main role of the PEP is to provide secure access to resources by validating trust. It consists of two main components, the `Policy_Decision_Point` (PDP) and the `PEP_Manager`. Whenever an access request appears at the PEP input, it is forwarded to the PDP via the `PEP_Manager`. The `Policy_Administrator` provides the interface between the PDP and `PEP_Manager`. This request is then evaluated by the `Policy_Engine` to validate trust using the rules defined by the `Policy_Enforcement_Point_Policy`. If necessary, the `Policy_Engine` further utilizes external information (labeled as `additional_info` in the figure) to validate trust. This can represent information such as analytics, decisions from other software tools, specific device-related information, etc. If the trust is validated, the PEP provides access to the requested resource; otherwise, the request is denied.

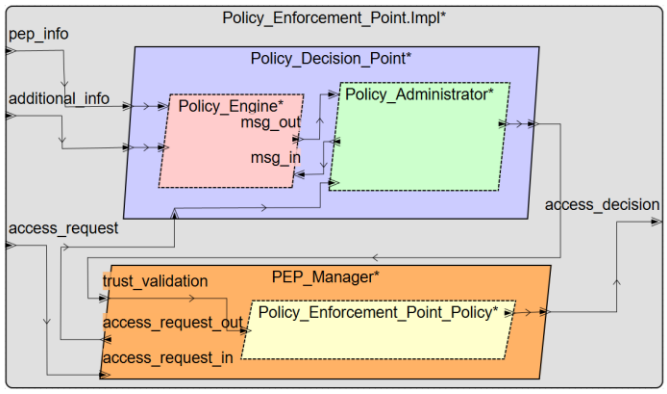

Figure 8. Architecture for the policy enforcement point mechanism.

Figure 9 shows the textual AADL representation of the PEP mechanism. First, we define the component type for the PEP. This consists of several features (lines 75-78) that represent various input/output ports. For the specification of the PEP in AADL, each PEP mechanism will consist of a `pep_info` port, which will contain the necessary information needed to validate trust. This information could come from another ZT mechanism, such as attestation or from any other valid source. Additional ports function as normal input/output ports connected internally by a switch mechanism such that information flowing into the component is only permitted to pass after trust is established. Because any additional ports only act as information flow ports, they are omitted from this PEP representation for the sake of clarity, however they do appear in the UAV example in Figure 26. Next, we define the PEP implementation that represents the sub-components (lines 85-86). Each of these sub-components are further defined in AADL.

In order to identify the PEP component in the model we have set a property as defined in line 80. This property will be useful during specific analyses that are performed on system models containing PEP components.

```
73  system Policy_Enforcement_Point
74      features
75          access_request: in event data port;
76          access_decision: out event data port;
77          additional_info: in event data port;
78          pep_info: in event data port;
79      properties
80          ZTA_Properties::Component_Type => POLICY_ENFORCEMENT_POINT;
81  end Policy_Enforcement_Point;
82
83  system implementation Policy_Enforcement_Point.Impl
84      subcomponents
85          Policy_Decision_Point: process Pol_Decision_Point.Impl;
86          PEP_Manager: process PEP_Mgr.Impl;
87      connections
88          c1: port additional_info -> Policy_Decision_Point.additional_info;
89          c2: port access_request -> PEP_Manager.access_request_in;
90          c3: port PEP_Manager.access_request_out -> Policy_Decision_Point.access_request;
91          c4: port Policy_Decision_Point.trust_validation -> PEP_Manager.trust_validation;
92          c5: port PEP_Manager.access_decision -> access_decision;
93          c6: port pep_info -> Policy_Decision_Point.pep_info;
94  end Policy_Enforcement_Point.Impl;
```

Figure 9. AADL textual representation of the PEP mechanism.

Figure 10 shows the `Policy_Decision_Point` and `PEP_Manager` sub-components of the PEP mechanism in AADL. Each of these components is defined using the component type (lines 36-42, and lines 57-63) and its implementation (lines 44-55, and lines 65-71). The overall design consists of the same elements that are described in Figure 8.

```
36  process Pol_Decision_Point
37      features
38          access_request: in event data port;
39          trust_validation: out event data port;
40          additional_info: in event data port;
41          pep_info: in event data port;
42  end Pol_Decision_Point;
43
44  process implementation Pol_Decision_Point.Impl
45      subcomponents
46          Policy_Engine: thread Pol_Engine.Impl;
47          Policy_Administrator: thread Pol_Admin.Impl;
48      connections
49          c1: port Policy_Engine.msg_out -> Policy_Administrator.msg_in;
50          c2: port Policy_Administrator.msg_out -> Policy_Engine.msg_in;
51          c3: port access_request -> Policy_Administrator.access_request;
52          c4: port Policy_Administrator.trust_validation -> trust_validation;
53          c5: port additional_info -> Policy_Engine.additional_info;
54          c6: port pep_info -> Policy_Engine.pep_info;
55  end Pol_Decision_Point.Impl;
56
57  process PEP_Mgr
58      features
59          access_request_in: in event data port;
60          access_request_out: out event data port;
61          trust_validation: in event data port;
62          access_decision: out event data port;
63  end PEP_Mgr;
64
65  process implementation PEP_Mgr.Impl
66      subcomponents
67          Policy_Enforcement_Point_Policy: thread PEP_Policy.Impl;
68      connections
69          c1: port trust_validation -> Policy_Enforcement_Point_Policy.trust_validation;
70          c2: port Policy_Enforcement_Point_Policy.access_decision -> access_decision;
71  end PEP_Mgr.Impl;
```

Figure 10. AADL textual representation of the PEP sub-components.

Figure 11 further represents the sub-components of the `Policy_Decision_Point` (lines 46-47) and `PEP_Manager` (line 67) components shown in Figure 10. Each of the sub-components represent their own component types and implementations that align with the architecture model of the PEP shown in Figure 8.

Run Time Integrity Monitor

Figure 12 represents a run-time integrity monitor mechanism. The role of this mechanism is to ensure that any abnormal system behavior is identified and flagged. The monitor observes a signal, which could be raw data from a sensor, or the contents of a data stream from another component. The monitor may contain a `Signal_Processing_Algorithm` to further process the observation signal. The `Monitor_Policy` then compares the observation with a reference value (`ref`) and will generate an alert if the observation


```
5   thread Pol_Engine
6       features
7           msg_in: in event data port;
8           msg_out: out event data port;
9           additional_info: in event data port;
10          pep_info: in event data port;
11  end Pol_Engine;
12
13  thread implementation Pol_Engine.Impl
14  end Pol_Engine.Impl;
15
16  thread Pol_Admin
17      features
18          msg_in: in event data port;
19          msg_out: out event data port;
20          access_request: in event data port;
21          trust_validation: out event data port;
22  end Pol_Admin;
23
24  thread implementation Pol_Admin.Impl
25  end Pol_Admin.Impl;
26
27  thread PEP_Policy
28      features
29          trust_validation: in event data port;
30          access_decision: out event data port;
31  end PEP_Policy;
32
33  thread implementation PEP_Policy.Impl
34  end PEP_Policy.Impl;
```

Figure 11. AADL textual representation of the PEP sub-components.

deviates by more than an acceptable threshold value. Otherwise, no alert is generated, indicating that the system is performing normally. Note that in this representation the reference is defined within the monitor, but it could just as well be another input to the component.

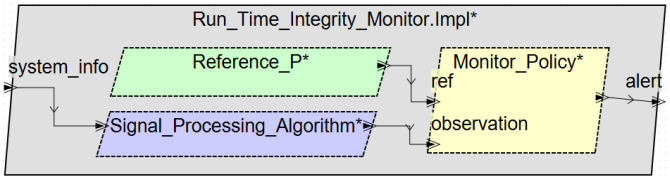

Figure 12. Architecture for run-time monitors.

Figure 13 shows the textual AADL representation of the run-time monitor mechanism. First, we define the component type of the run-time monitor element. It consists of various features (lines 30-31) representing the various input/output ports of the mechanism. The `alert` port (line 31) is defined to carry the system functionality information, i.e., normal or abnormal as a binary value. This information is computed based on the input present at the monitor's observation input port and compared against a pre-defined threshold value. Next, we define the implementation of the run-time monitor mechanism (lines 36-46). The sub-components of the run-time monitor are shown in lines 37-40, while their connections are represented using lines 42-45. The sub-components for the run-time monitor are defined in detail as threads (lines 4-26). In order to identify the run-time monitor component in the model we have set a property as defined on line 33. This property will be useful during specific analysis that can be performed on the system model containing the run-time monitor component.

```
4   thread Sig_Process_Algorithm
5       features
6           system_info: in event data port;
7           observation: out event data port;
8   end Sig_Process_Algorithm;
9   thread implementation Sig_Process_Algorithm.Impl
10  end Sig_Process_Algorithm.Impl;
11
12  thread Ref
13      features
14          ref: out event data port;
15  end Ref;
16  thread implementation Ref.Impl
17  end Ref.Impl;
18
19  thread Mon_Policy
20      features
21          observation: in event data port;
22          ref: in event data port;
23          alert: out event data port;
24  end Mon_Policy;
25  thread implementation Mon_Policy.Impl
26  end Mon_Policy.Impl;
27
28  process Run_Time_Integrity_Monitor
29      features
30          system_info: in event data port;
31          alert: out event data port;
32      properties
33          ZTA_Properties::Component_Type => MONITOR;
34  end Run_Time_Integrity_Monitor;
35
36  process implementation Run_Time_Integrity_Monitor.Impl
37      subcomponents
38          Signal_Processing_Algorithm: thread Sig_Process_Algorithm.Impl;
39          Reference_P: thread Ref.Impl;
40          Monitor_Policy: thread Mon_Policy.Impl;
41      connections
42          c1: port system_info -> Signal_Processing_Algorithm.system_info;
43          c2: port Reference_P.ref -> Monitor_Policy.ref;
44          c3: port Monitor_Policy.alert -> alert;
45          c4: port Signal_Processing_Algorithm.observation -> Monitor_Policy.observation;
46  end Run_Time_Integrity_Monitor.Impl;
```

Figure 13. AADL textual representation of the run-time monitor.

Trusted Data Load

Trusted data load ensures that data load integrity and authenticity is maintained. This consists of two components, a Trusted Data Load Verification process and a Trusted Data Load Signing process. Both components maintain data authenticity and integrity during their individual phase of the process. We describe each of these components in detail below.

Trusted Data Load Verification

Figure 14 shows the `Data_Load_Verification` process of the trusted data load mechanism. Input to the `Verification_Process` is signed software or data that needs to be transported to the target system. This information is passed to the `Data_Load_Verif_Manager`, which 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 (i.e., an organization that owns the software). In order to achieve this, it uses the security policies set in the `Data_Load_Verification_Policy`, which ensure the delivery or installation of verified software or data to the target system.

The textual AADL representation of the trusted data load verification mechanism is shown in Figure 15. First, we define the component type of the trusted data load verification mechanism (lines 61-67). This consists of features representing the input/output ports of the mechanism. The input port receives a signed data load, and the output port delivers the data load once verification is successful. Next, we

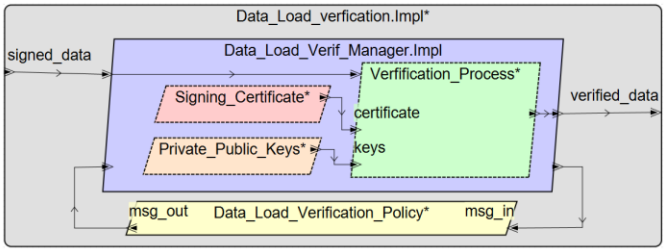

Figure 14. Trusted data load verification mechanism.

define the implementation of the trusted data load verification mechanism (lines 69-78). It represents the sub-components of the trusted data load verification mechanism (lines 70-72) and their connections (lines 73-77). In order to identify the trusted data load verification component in the model we have set a property as defined in line 66. This property will be useful during specific analyses that can be performed on the system model containing trusted data load verification components.

Figure 15 shows a representation of the `Data_Load_Verification_Manager` component type and its implementation. Other sub-components including sub-components of the `Data_Load_Verification_Manager` and their implementation is represented in Figure 16.

```
41  process Data_Load_Verif_Manager
42      features
43          signed_data: in event data port;
44          verified_data: out event data port;
45          msg_in: in event data port;
46          msg_out: out event data port;
47  end Data_Load_Verif_Manager;
48
49  process implementation Data_Load_Verif_Manager.Impl
50      subcomponents
51          Private_Public_Keys: thread Public_Private_Keys.Impl;
52          Signing_Certificate: thread Signing_Cert.Impl;
53          Verification_Process: thread Verf_Process.Impl;
54      connections
55          c1: port signed_data -> Verification_Process.signed_data;
56          c2: port Private_Public_Keys.keys -> Verification_Process.keys;
57          c3: port Signing_Certificate.certificate -> Verification_Process.certificate;
58          c4: port Verification_Process.verified_data -> verified_data;
59  end Data_Load_Verif_Manager.Impl;
60
61  system Data_Load_verfication
62      features
63          signed_data: in event data port;
64          verified_data: out event data port;
65      properties
66          ZTA_Properties::Component_Type => TRUSTED_DATA_LOAD_VERIFICATION;
67  end Data_Load_verfication;
68
69  system implementation Data_Load_verfication.Impl
70      subcomponents
71          Data_Load_Verification_Manager: process Data_Load_Verif_Manager.Impl;
72          Data_Load_Verification_Policy: process Data_Load_Verf_Policy.Impl;
73      connections
74          c1: port signed_data -> Data_Load_Verification_Manager.signed_data;
75          c2: port Data_Load_Verification_Manager.verified_data -> verified_data;
76          c3: port Data_Load_Verification_Manager.msg_out -> Data_Load_Verification_Policy.msg_in;
77          c4: port Data_Load_Verification_Policy.msg_out -> Data_Load_Verification_Manager.msg_in;
78  end Data_Load_verfication.Impl;
```

Figure 15. AADL textual representation of the trusted data load verification mechanism.

Trusted Data Load Signing

Figure 17 represents the architecture diagram for a `Data_Load_Signing` component. It is used to sign the software or data that needs to be delivered. This process maintains the data authenticity and integrity during its creation. It is similar to the `Data_Load_Verification` component. However, the only difference is that it uses a `Signing_Process` algorithm instead of a `Verification_Process` algorithm for data load. Once `unsigned_data` is available at the `Signing_Process` component of the `Data_Load_Sign_Manager`, it utilizes the same `Private_Public_Keys` and `Signing_Certificate`

to produce a `signed_data` ensuring it is signed by the trusted entity prior to its delivery.

```
5   thread Public_Private_Keys
6       features
7           keys: out event data port;
8   end Public_Private_Keys;
9
10  thread implementation Public_Private_Keys.Impl
11  end Public_Private_Keys.Impl;
12
13  thread Signing_Cert
14      features
15          certificate: out event data port;
16  end Signing_Cert;
17
18  thread implementation Signing_Cert.Impl
19  end Signing_Cert.Impl;
20
21  thread Verf_Process
22      features
23          keys: in event data port;
24          certificate: in event data port;
25          verified_data: out event data port;
26          signed_data: in event data port;
27  end Verf_Process;
28
29  thread implementation Verf_Process.Impl
30  end Verf_Process.Impl;
31
32  process Data_Load_Verf_Policy
33      features
34          msg_in: in event data port;
35          msg_out: out event data port;
36  end Data_Load_Verf_Policy;
37
38  process implementation Data_Load_Verf_Policy.Impl
39  end Data_Load_Verf_Policy.Impl;
```

Figure 16. AADL textual representation of trusted data load verification sub-components.

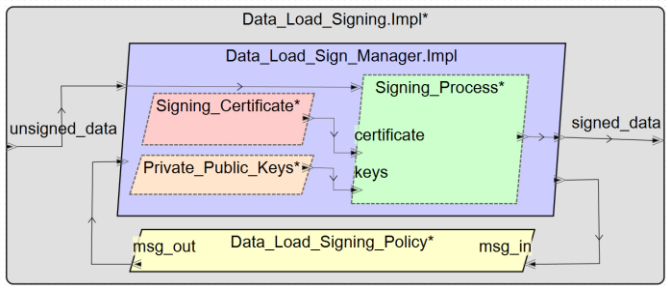

Figure 17. Trusted data load signing mechanism.

The textual AADL representation of the trusted data load signing component is same as the representation shown in Figure 15 and Figure 16. However, the only difference will be the representation of specific components that constitute the trusted data load signing component.

Separation Kernel

Figure 18 represents the design for a separation kernel mechanism. It ensures time and space partitioning while maintaining the integrity of applications. The separation kernel is a unique mechanism as it adheres to several ZTA tenets inherently [12]. Each process (`Process_i`) is bound to a processor (`Proc`) which hosts an operating system that provides separation guarantees. Processes contain a single thread, representing partitioning in both time and space. Further, each process is bound to a specific address space in the memory (`Mem`). Components communicate with each other over a hardware bus (`Bus_HW`). Additional bus bindings have been hidden for clarity. When an application is executed, it runs within its own process in an isolated address space. Simultaneous applications execute within their individual processes and memory space without interfering with other applications. This ensures time and space partitioning while maintaining application integrity.

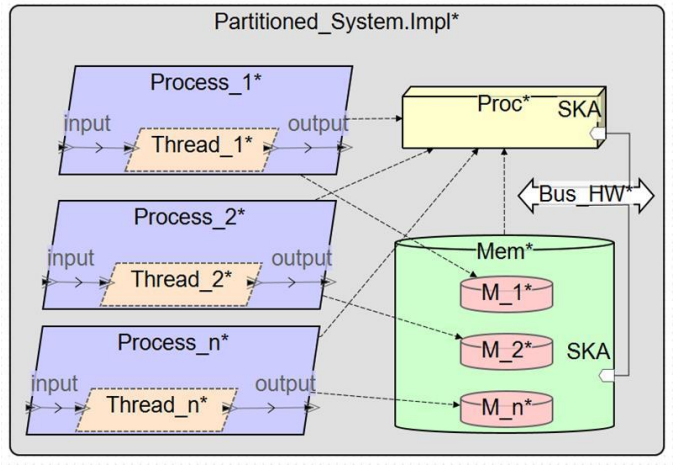

Figure 18. Architecture for the separation kernel mechanism.

```
106 memory M
107     features
108         SKA: requires bus access Bus_HW.Impl;
109 end M;
110
111 memory implementation M.Impl
112     subcomponents
113         M_1: memory m_1.Impl;
114         M_2: memory m_2.Impl;
115         M_n: memory m_n.Impl;
116 end M.Impl;
117
118 system Partitioned_System
119 end Partitioned_System;
120
121 system implementation Partitioned_System.Impl
122     subcomponents
123         Bus_HW: bus Bus_HW.Impl;
124         Process_1: process process_1.Impl;
125         Process_2: process process_2.Impl;
126         Process_n: process process_n.Impl;
127         Proc: processor P.Impl;
128         Mem: memory M.Impl;
129     connections
130         bac1: bus access Proc.SKA <-> Bus_HW;
131         bac2: bus access Mem.SKA <-> Bus_HW;
132     properties
133         ZTA_Properties::Component_Type => SEPARATION_KERNEL applies to Proc;
134         Actual_Processor_Binding => (reference (Proc)) applies to Process_1,
135             Process_2, Process_n;
136         Actual_Memory_Binding => (reference (Mem.M_1)) applies to Process_1;
137         Actual_Memory_Binding => (reference (Mem.M_2)) applies to Process_2;
138         Actual_Memory_Binding => (reference (Mem.M_n)) applies to Process_n;
139         Actual_Memory_Binding => (reference (Proc)) applies to Mem;
140 end Partitioned_System.Impl;
```

Figure 19. AADL textual representation of the separation kernel mechanism

Figure 19 shows the textual representation of the separation kernel pattern in AADL. The system implementation is represented in lines 121-129, along with its sub-components (lines 122-128), their connections (lines 129-131), and associated properties (lines 131-138). The sub-components are defined with their component types and their respective implementations in Figure 19 and Figure 20. Figure 19 represents the memory type and its implementation (lines

106-116) whereas Figure 20 shows the bus, process, and processor component types and implementations (lines 60-92). Note that only one component type and implementation of each component is shown in Figure 20 for the sake of brevity. The processor and memory bindings are represented in Figure 19 (lines 134-138). Line 134 defines the processor binding to each of the process. Further, lines 135-137 shows the binding between each process with individual memory space within the memory (`Mem`). Line 138 shows the binding between the processor (`Proc`) and the memory (`Mem`). In addition, to identify the separation kernel component in the model, we have set a property as defined in line 133. This property will be useful during specific analysis that can be performed on the system model containing separation kernel component.

```
60  bus Bus_HW
61  end Bus_HW;
62
63  bus implementation Bus_HW.Impl
64  end Bus_HW.Impl;
65
66  process process_n
67      features
68          input: in event data port;
69          output: out event data port;
70  end process_n;
71
72  process implementation process_n.Impl
73      subcomponents
74          Thread_n: thread thread_n.Impl;
75      connections
76          c1: port input -> Thread_n.input;
77          c2: port Thread_n.output -> output;
78  end process_n.Impl;
79
80  processor P
81      features
82          SKA: requires bus access Bus_HW.Impl;
83  end P;
84
85  processor implementation P.Impl
86  end P.Impl;
87
88  memory m_1
89  end m_1;
90
91  memory implementation m_1.Impl
92  end m_1.Impl;
```

Figure 20. AADL textual representation of separation kernel mechanism sub-components.

Filter

Figure 21 shows the architecture diagram for the filter mechanism. The purpose of a filter component is to allow only ZTA-compliant inputs to propagate. The definition of compliance can be defined as a security policy inside the `Filter_Policy` component. Whenever an input arrives at its ports, compliance is checked using the defined rules. The `Filter_Algorithm`, along with the `Filter_Policy`, perform this compliance check. Input data is then allowed to be placed on its output port if a successful compliance is achieved; otherwise, the input is dropped. Filter policies can easily be customized depending upon the type of input.

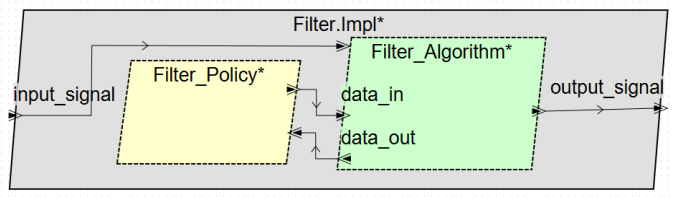

Figure 21. Architecture for the filter mechanism.

Figure 22 shows the textual AADL representation of the filter mechanism that follows the design discussed above. First, we define the component type of the filter mechanism in AADL (lines 25-31). This consists of various input/output ports (lines 27-28). Next, we define the implementation of the filter mechanism (lines 33-42). This consists of `Filter_Algorithm` and `Filter_Policy` sub-components (line 34-36) and the connections associated with them (lines 37-41). The `Filter_Algorithm` and `Filter_Policy` threads are defined using their own component type and implementation (lines 5-23). In order to identify the filter component in the model, we have set a property as defined in line 30. This property will be useful during specific analysis that can be performed on the system model containing filter component.

```
5   thread Filter_Algo
6       features
7           input_signal: in event data port;
8           output_signal: out event data port;
9           data_out: out event data port;
10          data_in: in event data port;
11  end Filter_Algo;
12
13  thread implementation Filter_Algo.Impl
14  end Filter_Algo.Impl;
15
16  thread Filt_Policy
17      features
18          data_in: in event data port;
19          data_out: out event data port;
20  end Filt_Policy;
21
22  thread implementation Filt_Policy.Impl
23  end Filt_Policy.Impl;
24
25  process Filter
26      features
27          input_signal: in event data port;
28          output_signal: out event data port;
29      properties
30          ZTA_Properties::Component_Type => FILTER;
31  end Filter;
32
33  process implementation Filter.Impl
34      subcomponents
35          Filter_Algorithm: thread Filter_Algo.Impl;
36          Filter_Policy: thread Filt_Policy.Impl;
37      connections
38          c1: port input_signal -> Filter_Algorithm.input_signal;
39          c2: port Filter_Algorithm.output_signal -> output_signal;
40          c3: port Filter_Algorithm.data_out -> Filter_Policy.data_in;
41          c4: port Filter_Policy.data_out -> Filter_Algorithm.data_in;
42  end Filter.Impl;
```

Figure 22. AADL textual representation of the filter.

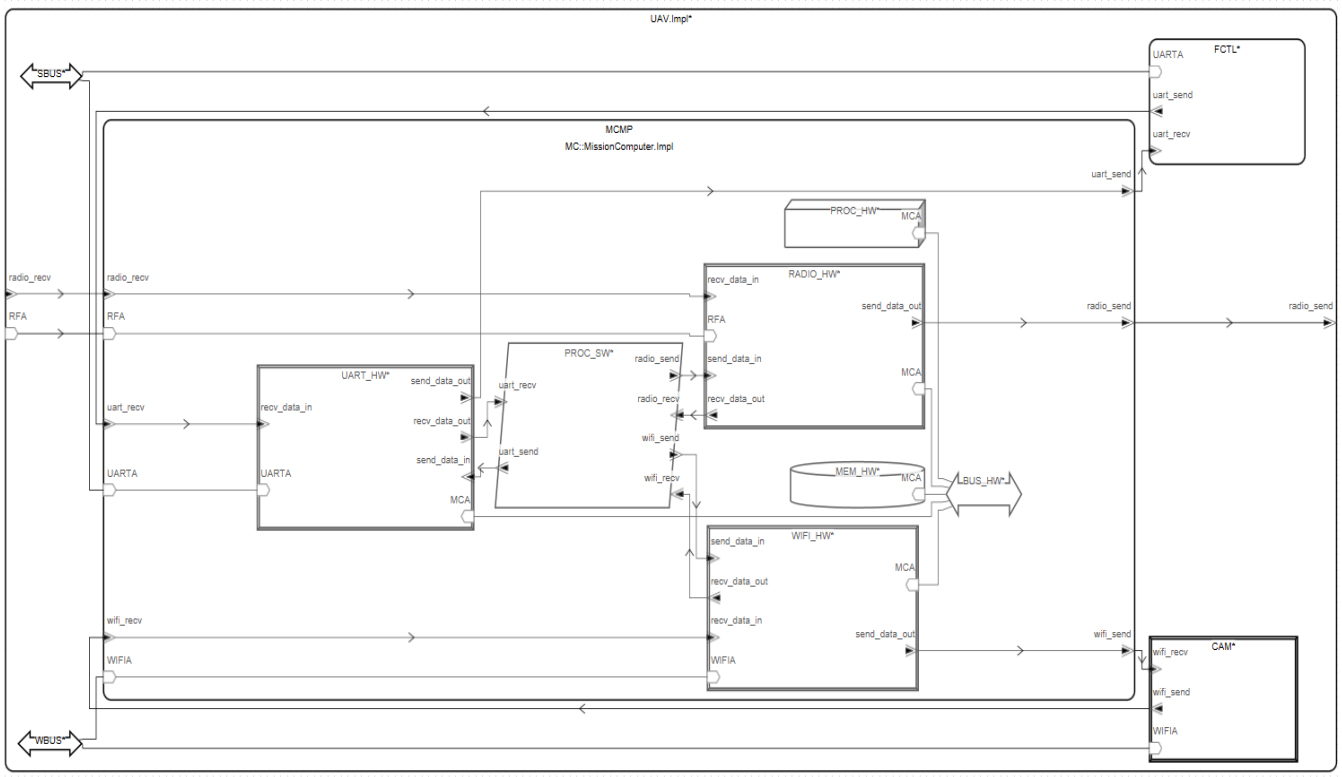

Figure 23. Initial UAV model in AADL with no ZT mechanisms.

Experimental Model and ZT Application

In this section, we will first describe our use case experimental model. Next, we will demostrate the application of the ZT mechanisms defined in the previous section to the use case model. In addition, we will discuss the advantages of the applied ZT mechanisms in terms of improving the system security posture.

Initial UAV Architecture with no ZT Mechanisms

We have considered a UAV system for our case study. The function of this UAV is to carry out automated surveillance missions within some pre-determined geographical area. Figure 23 shows the UAV architecture in AADL. The model consists of a mission computer, flight controller, camera, and physical buses for data/message exchange. Further, the mission computer consists of the hardware devices associated with the UAV, i.e., radio, WiFi, camera hardware, etc. It also consists of the software module responsible for the behavior and functionality of the UAV system. The flight controller consists of the GPS device for navigation purposes and to capture the UAV's position. In addition, the model also consists of AADL elements for the system processor and memory. This initial model does not contain any ZT mechanism discussed above, and hence the system remains vulnerable to a variety of cyber threats. For instance, the UAV may need to communicate with a ground station to receive updates on its mission plans. However, since there is no mechanism to validate the integrity of the software running on the ground station, a cyber vulnerability exists that an attacker can exploit. An adversary can easily ingest malicious software into the ground station code base and start communicating with the UAV. Once communication is established, the adversary could then pose a severe threat to the health of the UAV and the succes of the overall mission, (for example, by supplying false mission plans that lead the UAV to crash or get captured). Other cyber vulnerabilities can result in similar critical mission failures.

Initial UAV Software Architecture with no ZT Mechanisms

Figure 24 represents the UAV software architecture. The software resides on the mission computer of the UAV. It consists of various device drivers such as radio, WiFi, UART, etc. Further, it also consists of the sub-components that are responsible for capturing the application logic such as the waypoint manager, flight planner, camera manager, and no fly zone database. Since this design of the UAV software architecture does not contain any ZT mechanisms, it remains vulnerable to various cyber threats. For instance, when the UAV receives a mission command over the radio from the ground station, the message is directed to the flight planner component (`FLPN`) by the radio driver. The flight planner will use this information to generate a new flight plan, and send the updates to other modules such as camera manager (`CM`) and waypoint manager (`WPM`), which will perform their own computations to assist the UAV in executing the mission. However, several security threats can result from this design. For example, in the absence of an attestation mechanism, the UAV could be communicating with a legitimate ground station running corrupt software, and thus potentially receive false mission commands. The possibility also exists that the mission command received by the UAV is malformed even though the UAV is communicating with a legitimate ground station running authentic software. Both of these scenarios can easily result in mission failure and ultimately lead to catastrophic consequences.

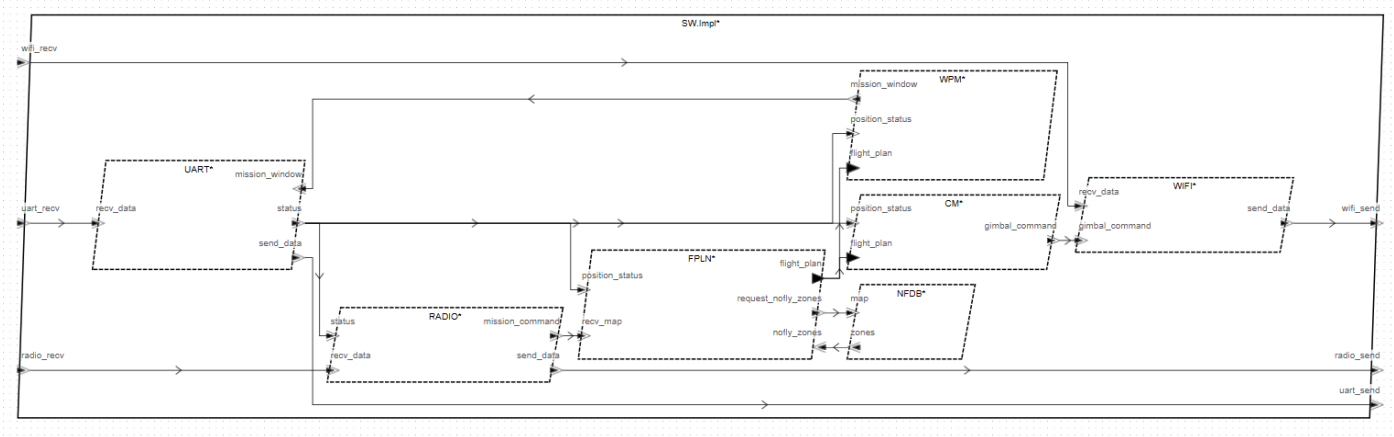

Figure 24. Initial UAV software architecture in AADL with no ZT mechanisms.

Hardened UAV Architecture with ZT Mechanisms

To improve the security posture of the UAV model we have applied the ZT mechanisms discussed above. Figure 25 shows the hardened UAV model with the applied ZT mechanisms. The UAV consists of the same ZT components as discussed above. We modeled the encryption mechanism at this layer of the model because all messages are directed to and from this level to their respective destinations. By employing the encrytion mechanism, we can argue that the messages flowing on the associated connections will remain secure and encrypted, thereby avoiding the possibility of eavesdropping by adversarial entities. Other ZT mechanisms are applied at the software component level of the UAV model, described next.

Hardened UAV Software Architecture with ZT Mechanisms

Figure 26 represnts the UAV software architecture with the application of ZT mechanisms. In addition to the applied ZT mechanisms, it contains all the components from the initial model. As can be seen in the zoomed inset of the figure, the updated software consists of ZT mechanisms such as attestation, policy enforcement point, filter, and trusted data load. Now, we consider the same scenario as discussed in the initial software component with no ZT mechanisms. If there is a radio signal received by the UAV and it is present at the radio device driver (`Radio`), it will not allow this signal to pass through until it checks the integrity of the source, i.e., ground station software. To achieve this, the capabilities of the attestation mechanism are utilized. First, the attestation mechanism initiates an attestation request, and receives a response that consists of the necessary information to perform attestation. Using this information, attestation performs the software integrity evaluation of the ground station and produces the necessary information at its `Policy_Info` port. Next, this information is utilized by the PEP to determine whether to allow communication between the `Radio` driver and other components in the system. The critical outputs of the `Radio` driver are only allowed to become available at the respective components in the system only if the PEP establishes the necessary trust. This process validates the integrity of the ground station software and eliminates the cyber threats resulting from lack of attestation. In addition, we have inserted a filter mechanism between the PEP and the `Flight Planner` component. Once the PEP passes the mission command to the filter mechanism, it checks to determine whether the command is malformed. This prevents the UAV from performing any illegal mission operations based on malformed commands, which could result in damage to the UAV and failure of the mission.

As another example, let us consider the scenario for a software update over the radio for the UAV, assuming that the software integrity of the source is already established by the attestation mechanism and the PEP has established the trust between the `Radio` driver and the `Trusted_Data_Load` components. Upon receiving the updates from the PEP, the `Trusted_Data_Load` will validate the authenticity and integrity of the software based on the mechanism described in above. It will only allow the updates to reach the `Software_Updater` module if it determines that the software originated from a valid entity and the integrity of the software is maintained. The `Software_Updater` will then perform the updates. If such a mecahnism is not present, any arbitrary software update can be performed, which could result in severe consequences. Utilizing the defined ZT mecahnisms will therefore result in an improved system security posture while minimizing various cyber vulnerabilities.

Conclusions and Future Work

Security for CPS is becoming an important challenge. Zero trust is an emerging technology that has proven very effective in addressing security in the IT infrastructure domain. We have identified ZT architecture patterns targeted specifically to CPS. We defined these patterns in AADL and made them available to engineers as a library of ZT components. With appropriate tool automation, these components can be inserted into an existing AADL model in order to provide the security enhancements associated with the ZT components. We have demonstrated our approach using a UAV surveillance system and discussed the use of our ZT patterns to improve overall system security.

In future work, we will further develop our approach to build a tool that provides the ability to leverage ZT architecture patterns and build ZT compliant CPS systems while providing design-time assurance that the system design is indeed ZT compliant. Engineers can easily use the design assurance capability to build ZT compliant systems while supporting their overall system design requirements. In addition, this process will identify vulnerabilities early in design and will enable engineers to take appropriate action to mitigate them.

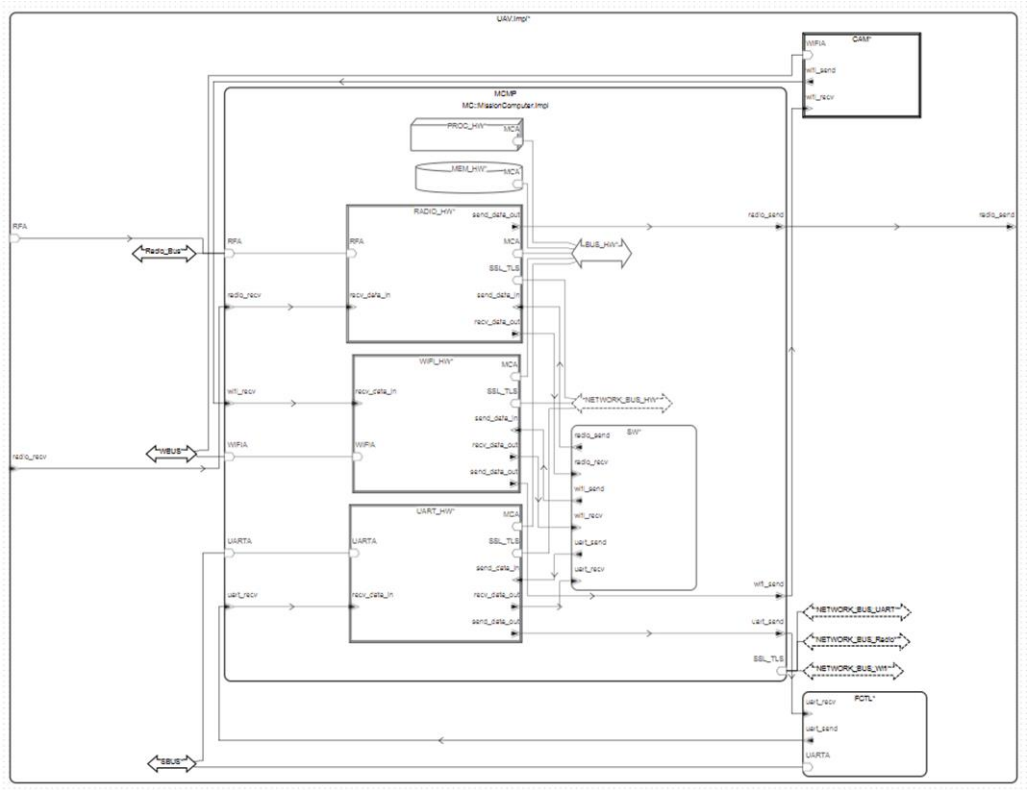

Figure 25. Hardened UAV model in AADL with added ZT mechanisms.

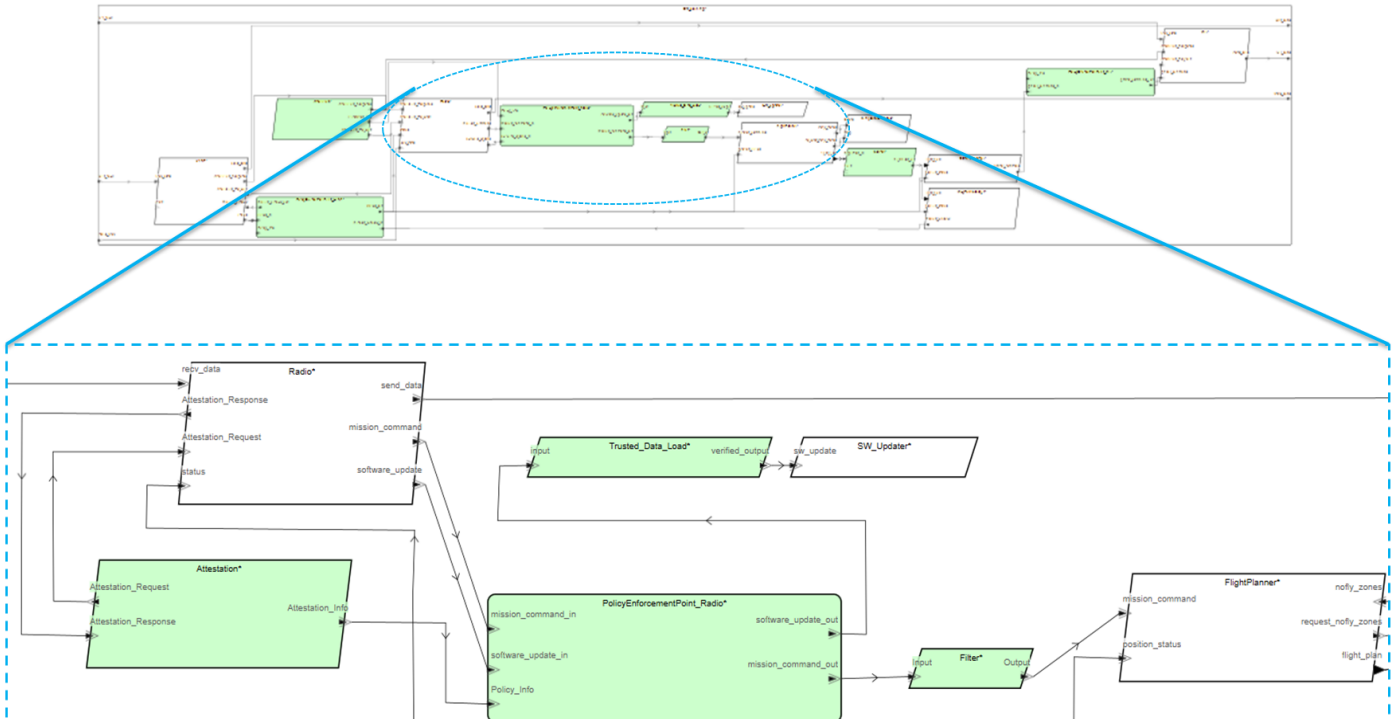

Figure 26. Hardened software architecture of the UAV model in AADL with ZT mechanisms.

References

1. "Cars face cyber threats too," The Washington Post, accessed November 8, 2022, https://www.washingtonpost.com/politics/2022/10/18/cars-face-cyber-threats-too/.
2. "Aviation is facing a rising wave of cyber-attacks in the wake of COVID," Stephenson Harwood, accessed November 8, 2022, https://www.shlegal.com/insights/aviation-is-facing-a-rising-wave-of-cyber-attacks-in-the-wake-of-covid.
3. Freter, R., "Department of Defense (DOD) Zero Trust Reference Architecture," accessed November 12, 2022, https://dodcio.defense.gov/Portals/0/Documents/Library/(U)ZT_RA_v2.0(U)_Sep22.pdf.
4. Kerman, A., Olive, B., Scott, R., and Allen, T., "Implementing a zero trust architecture," National Institute of Standards and Technology 2020 (2020): 17-17.
5. Walker-Roberts, S., Mohammad, H., Omar, A., Mehmet, A., and Ali, D., "Threats on the horizon: Understanding security threats in the era of cyber-physical systems," The Journal of Supercomputing 76, no. 4 (2020): 2643-2664.
6. Conlon, C., and Cesare, G., "A New Zero-Trust Model for Securing Embedded Systems," In Proceedings of the Embedded World Conference, Nuremberg, Germany. 2019.
7. Papakonstantinou, N., Douglas, L., Joona, L., et. al., "A zero trust hybrid security and safety risk analysis method," Journal of Computing and Information Science in Engineering 21, no. 5 (2021).
8. Feiler, P., David G., and John H., "The architecture analysis & design language (AADL): An introduction," Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, 2006.
9. Feiler, P., and David G., "Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language," Addison-Wesley, 2012.
10. SEI AADL Team, "OSATE: Plug-ins for front-end processing of AADL models, 2008," http://la.sei.cmu.edu/aadl/currentsite/tool/osate. html.
11. Cofer, D., Amundson, I., Babar, J., Hardin, D., et. al. "Cyber Assured Systems Engineering at Scale," IEEE Security and Privacy, May-June 2022.
12. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., et. al. "seL4: formal verification of an OS kernel," Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009 (SOSP 2009). Big Sky, Montana, USA., October 11-14, 2009. J. N. Matthews and T. E. Anderson, Eds. ACM, 2009. pp. 207–220.
13. Amundson, I., and Cofer, D., "Resolute assurance arguments for cyber assured systems engineering." In Proceedings of the Workshop on Design Automation for CPS and IoT, pp. 7-12. 2021.

Contact Information

Saqib Hasan is a Senior Research Engineer at Collins Aerospace. He can be reached at saqib.hasan@collins.com.

Acknowledgement

The authors would like to thank the reviewers for providing constructive feedback on the paper.