An Architecture-Centric Process for MILS Development

Julien Delange, Min-Young Nam, Peter Feiler, and Will Klieber
Carnegie Mellon Software Engineering Institute
4500 Fifth Avenue
Pittsburgh, PA 15213-2612, USA
{jdelange,mnam,phf,weklieber}@sei.cmu.edu

ABSTRACT

Safety-critical embedded systems are now software-reliant and evolving at an incredible pace. With the emerging Internet of Things (IoT) ecosystem, these systems are now interconnected to several networks and exposed to potential attackers. This increases the potential surface of attack and, ultimately, the likelihood of a successful attack that would penetrate the system. Until recently, many security efforts were focused on code analysis, but studies have shown that security is also a matter of good software architecture design and practices. For example, MILS requires isolating security domains in partitions using appropriate security components. However, because embedded systems are evolving quickly, new design methods are now required to overcome the challenges of developing them.

In this paper, we introduce a research agenda for a new architecture-centric development approach for MILS systems. This would leverage architecture models and augment them with security information in order to perform the different activities of the development process, including security policy validation, implementation, and testing. Using the same model throughout development improves the consistency of the development process by avoiding any translation between different—and potentially inconsistent—representations. In addition, automating the generation of implementation and tests avoids the traditional mistakes of manual code production, such as bugs and developers' assumptions about ambiguous requirements.

Categories and Subject Descriptors

D.4.6 [**Operating System**]: Security and Protection—*Verification*; D.2.11 [**Software Engineering**]: Software Architecture—*Languages*; D2.4 [**Software Engineering**]: Software Verification—*Validation*

General Terms

Security, MILS, Software Architecture, Assurance, Testing

Keywords

AADL, MILS, Security

MILS workshop '2016

1. INTRODUCTION

Safety-critical embedded systems now rely on software to a great degree and are no longer physically isolated as they used to be. Existing functions, previously implemented using hardware, are now realized using software. New features are also automatically implemented using software on different processors interconnected with specific buses. For example, automotive systems now contain a lot of interconnected functions classified at different criticality levels, such as low criticality for entertainment systems and high criticality for cruise control.

Because these systems are classified at different criticality levels, engineers must ensure that low-criticality functions cannot disturb functions of higher criticality (e.g., the entertainment system cannot disturb the cruise control). In addition, there has been an increasing demand to connect these systems to the Internet, making them available to the so-called Internet of Things ecosystem but also exposing them to external threats. The attack surface is no longer limited to physical system boundaries, and these critical systems must be protected from external threats. One common example is the emerging technology of self-driving cars, which are interconnected with noncritical (entertainment features) and critical (traffic control) infrastructures while operating a life-critical function (driving a car). These systems must be carefully designed to ensure that noncritical functions cannot interfere or disrupt critical ones (e.g., interrupting the driving function, which might result in a car accident).

Unfortunately, ensuring software isolation within the car or the absence of impact from external attack is very hard to achieve [1, 11, 12]. Over the last few years, researchers have demonstrated that actual systems are exposed to security attacks that could have catastrophic consequences; for example, an attacker could take full control of the car using a remote connection [12]). Such attacks have catastrophic consequences not only for people in the vehicle but also for manufacturers. A single successful attack requires the manufacturer to update the software and potentially recall all affected cars. In fact, in 2015 manufacturers recalled more than 1.8M cars—at least 1.4M for Chrysler and 433K for Ford—because of security-related software issues.

For years, security issues have been classified as coding issues (e.g., buffer overflow, inappropriate libraries). However, a better software architecture design, such as one that isolates software functions from each other from the beginning, would automatically block such attacks. While current research efforts focus on such approaches, they lack methods and tools to support them.

In the current paper, we present our agenda for a new approach to design secure systems using architecture models. Our methods aim to achieve the following objectives:

1. **Specify a software architecture with its security constraints** (security domains, security mechanisms, etc). We extend the Architecture Analysis and Design Language (AADL) [14] with a new annex dedicated to security modeling.

2. **Check the security policy correctness** with new dedicated analysis tools that process a model that has been annotated with security constraints. The tools validate the security architecture, detect issues, and suggest potential design improvements of the software architecture.

3. **Produce a security attack surface and an attack tree** from the software architecture model. This dedicated analysis tool processes security-enhanced AADL models [14] and generates documents that contain the attack surface as well as the attack tree of the system.

4. **Generate security attack tests** from the software architecture model. The attack tests will then stress-test the software architecture, to try to attack and challenge the software implementation. Such tests demonstrate that the security policy is correctly implemented. Even if our approach generates the security policy (see below), generating tests is still relevant, especially for certification purposes or when some parts of the system are manually implemented.

5. **Generate the security policy** using dedicated tools that auto-produce configuration files (e.g., XML files that configure dedicated embedded kernels or specific networks) that implement the security policy. This approach ensures that the security policy designed in the models and previously validated is correctly implemented.

In the following sections, we present current state-of-the-art research efforts, distinguish our contribution, and detail our research agenda for the next two years to realize these objectives.

2. OVERVIEW OF THE APPROACH

In this section, we first present existing related work. We then detail what is new in our approach. The next section explains each part of our approach.

2.1 Related Work

The MILS[1] principles were proposed by John Rushby years ago [13]. The overall goal is to isolate activities in different domains in distinct partitions. In a MILS system, activities in one partition must not impact activities in other partitions [3] unless there is a functional justification to do so. The isolation is realized in terms of space (software with different security domains should not exchange, or read/write, data) and time (each software partition has a dedicated, fixed amount of time to execute).

A MILS system relies on trustworthy seperation kernel and middleware to ensure isolation across partitions. Both commercial and open-source MILS separation kernels are currently available.

Hansson et al. [7] introduce modeling guidelines to map several MILS principles to the AADL [14]. This approach focused on the functional aspect of the system and neglected the execution platform (how the software is deployed), which can be a major threat if, for example, there is lack of encryption or no isolation between the execution of software in different security domains. In Delange et al. [6], the authors extend the approach to include deployment aspects and generate the security policy of the underlying MILS separation kernel.

Van der Pol et al. [17] proposed a MILS-specific notation (MILS-AADL) for the specification of MILS architecture. The notation contains necessary artifacts to define the system components with the characteristics of the security policy, such as protection mechanisms and security domains. Hawkins et al. [8] designed a method to use this notation to produce assurance cases and show the degree of compliance of a given architecture with MILS requirements.

2.2 Roadmap

Our approach relies on a new AADL security annex that is currently under definition. This new annex will provide the ability to capture security constraints in AADL models [14]. Functional aspects have been already addressed [6, 7, 17], but there has been limited support for validation of the association with the execution platform [6, 17]. The MILS-AADL language [17] reuses AADL principles and provides the ability to specify several security mechanisms, but it has two shortcomings: (1) some mechanisms, such as time and space isolation used in safety-critical operating systems, are not available, and (2) it is a new language and incompatible with the existing AADL standard, so models designed with MILS-AADL cannot be used with other analysis tools.

Our proposed new annex will capture security concerns in the architecture in a manner that is fully compatible with the core AADL standard [14]. It will allow designers to reuse models previously developed for other purposes, and extend them with security-specific modeling patterns (e.g., by capturing platform-specific security mechanisms such as time and space isolation of the MILS separation kernel). We are planning to submit a draft of the annex to the SAE-AADL standardization committee in 2016 and are planning to seek committee approval for standardization in 2017.

Validation tools will leverage this new notation and check security policy using augmented architecture models. For example, it will check that a components having different security domains are isolated and that legitimate communication use isolation mechanisms. Our tools will also produce the Attack Surface [10] and Attack Tree [16] from the models to support system validation efforts. We are currently working on these aspects and will release a first draft of these tools in early 2016.

The security policy of the system will be automatically generated from validated models. We will modify the Ocarina AADL toolsuite [18] in order to be able to generate the security policy from models (such as the separation kernel validation). We will release our modification in 2016 and plan to support mostly open-source software such as SeL4 [9] or POK [6] (the list of supported platforms still has to be decided).

Finally, security tests will be automatically generated from the models. This will help engineers to verify that the implementation of the desired MILS security policy is correct. Generating security tests from models could be useful in several contexts:

1. When certifying the system, such tests are required. As embedded systems implement more critical functions and enforcing security policies becomes more difficult, regulatory agencies will require evidence the system enforces the security policy. The security tests provide this evidence.

2. If the security platform is not certified or verified, security tests can detect potential issues in their implementation.

3. When configuration code is manually implemented and not

[1]Historically MILS stands for "Multiple Independent Levels of Security" and today is considered as a proper noun

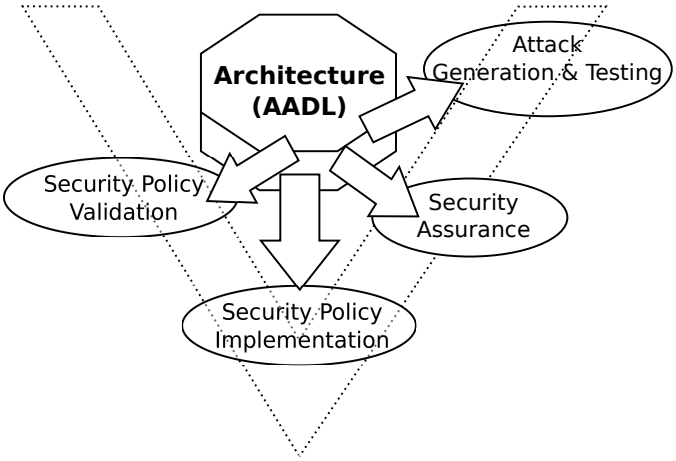

Figure 1: Integration of our approach in a traditional development process

generated from models, automating security tests from models will ensure that the engineers have correctly implemented the security policy and have not introduced any errors.

We will also implement the generation of security tests by extending the Ocarina AADL toolsuite [18] and its associated C code generator. Our modifications will be published in 2016.

3. DETAILS ABOUT THE APPROACH

Our approach uses an architecture specification enhanced with security annotation to drive several aspects of the development process. As illustrated in Figure 1, the annotated software architecture is used to:

1. validate the security policy at the design-level
2. implement the system and its security policy from models (e.g. generation of MILS components such as the separation kernel).
3. generate assurance of security policy conformance (e.g. assurance case)
4. generate security tests (e.g. attack of partitions separation) in order to check the correct implementation of the security policy.

This section details our approach and our plans to implement it.

3.1 Architecture Specification

The AADL [14] has been used for design and validation of safety-critical systems from different perspectives, such as performance [5] or safety [4]. The core language can be extended with user-defined properties and the language annex.

We propose a new security-specific annex to augment AADL models and define security constraints and requirements in architecture models. We used a similar approach for safety [15]; now our objective is to provide an extension from a security perspective.

This AADL annex aims at augmenting the core language to specify

- **security policy**: security domains and levels on components and their relationships
- **protection mechanisms**: isolation and protection features, such as the MILS separation kernel, encryption mechanisms of data, and security-specific protocols and buses
- **architecture annotation**: specification for components that have been formally verified and analyzed (e.g., a Multiple Levels of Security (MLS) component for filtering classified data to be sent to an unclassified network)

Figure 2 shows the functional view of a software architecture with two sender components, two receivers, and one merger. Each function is captured using an AADL `abstract` component (dashed rectangles on Figure 2). AADL `abstract` components can be later refined in specialized types, such as `system` or `process`. Users can then refine the functional architecture into an implementation architecture by adding new components, refining the existing ones, and preserving all the characteristics of the functional architecture.

In the following example, data are classified at two security domains: secret (in red) and unclassified (in green). The merger component is an MLS component that handles data at different security domains:

- secret data is encrypted by the merger and sent to another secret partition that will ultimately decrypt it
- unclassified data is separated and sent to an unclassified partition

Assuming that the `merger` component is marked as validated and verified, each security level is separated in a single component.

Because this MLS component handles data at different security levels, it should be verified and, eventually, annotated as being reviewed and validated for use. The AADL security annex will provide the ability to add this annotation.

The implementation and deployment of this functional architecture are shown in Figure 3. The implementation model adds the execution platform, represented using an AADL `processor` component (shown as a box with an unbroken line), and its partitions, represented using AADL `virtual processor` components (shown as boxes with dashed lines).

All functions (AADL `abstract` components) associated with a given security level are bound to a single partition. This association, which the AADL notation calls a *binding*, is represented using an arrow with a dashed line. The `merger` partition is associated with a dedicated partition. This implementation ensures the correct implementation of the security policies: each security level, as well as the MLS component, is confined in a single partition.

With this notation, it is then possible to (1) describe the security mechanisms to protect data flows (e.g., encryption) and (2) specify the security mechanisms of the execution runtime (e.g., time and space isolation as prescribed by MILS [2, 13]).

This new AADL annex will leverage existing research efforts [6, 17], and we will propose it as a new SAE standard to constitute the official AADL annotation mechanism to specify security concerns. We will also support this new AADL annex with a textual editor integrated in the Eclipse-based Open Source AADL Tool Environment (OSATE) tool so that model designers can annotate their models with an efficient textual editor.

3.2 Validating MILS Security Policy

Analysis tools process the AADL model with security annotations, including security policy and constraints, and check the model against the security policy to assess compliance. The tool works in two steps:

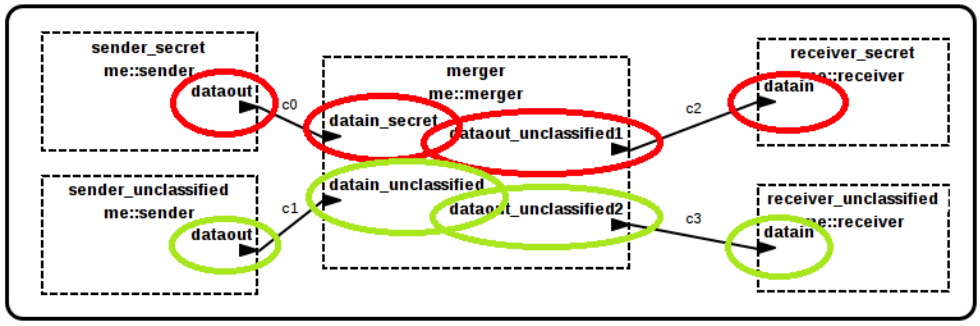

Figure 2: Functional view of the system

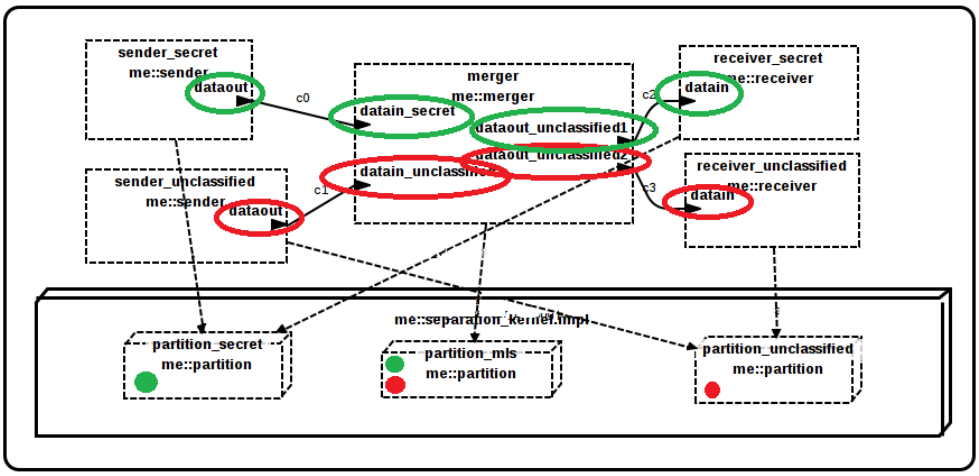

Figure 3: Implementation of the system

1. **Validating security policy**: Components' definitions and connections are compliant with the security policy. In this step, the tool checks that components do not mix security levels or downgrade data (unless they have been verified).

2. **Implementing correctness validation**: Implementation and allocation of components to the execution platform do not break the previously validated security policy. For example, when allocating two functions with different security levels that are assigned to the same processor, the analysis tool will check that appropriate security mechanisms (e.g., separation kernel) are used to ensure data isolation.

3.3 Generating the Attack Surface and Attack Tree

The same architecture model can be leveraged to generate the attack surface [10] and the attack tree [16].

The attack surface is an enumerated list of all potential attacks that the system is exposed to. The software architecture exhibits how functions are deployed on the execution platform. An analysis tool can then detect architectural defects that expose the system to security threats.

Similarly, an analysis tool can generate an attack tree. An attack tree is a comprehensive diagram that shows how a system can be compromised by listing all the related potential attacks and showing their dependencies using a tree layout. The system architecture notation, with its associated hierarchical nature, can be leveraged to produce an attack tree.

3.4 Generating Security Tests

An implementation architecture model, like the one shown in Figure 3, is used to produce comprehensive and application-specific security tests. With the security constraints in the model, tools can produce efficient tests, specifically tailored to the system being inspected.

For our example, the tool would create security tests to be executed in the unclassified partition that will try to read data stored in the secret partition. This would test the correct implementation of the MILS security policy and its associated space isolation across partitions. Similarly, for a distributed system, a test would check that appropriate encryption protocols are used to transport secret data, especially when the network is not physically isolated.

3.5 Implementing Security Policy

Finally, the architecture model and its security and implementation details are leveraged to generate the security policy [6]. This creates code (such as XML code) to configure MILS execution components (e.g., separation kernel, middleware) and security mechanisms (e.g., encryption algorithms, encryption key management).

In our example, the tool would (1) configure the separation kernel partitions, (2) deploy the encryption keys on appropriate components — the `merger` and `receiver_secret` — and (3) configure encryption algorithms to use the correct keys. Automating configuration and deployment code from a previously validated model ensures the correctness of the security policy implementation and avoids errors related to manual development efforts.

4. CONCLUSION AND NEXT STEPS

In this paper, we present our research agenda toward a new architecture centric design approach for MILS. Leveraging existing research efforts, our approach uses the software architecture as the main asset to validate the MILS isolation principles, configure the runtime components, show assurance security correctness, and, ultimately, test the system. Such an approach avoids the typical problems of producing code by hand and, by using the same model, guarantees the overall development process. It guarantees security policy correctness in the architecture and, coupled with code analysis tools that catch security issues at the code level, will significantly reduce potential security threats.

We will propose the developed security extension for AADL to the SAE AADL Subcommittee in order for it to become the official security annotation for this architecture language. New tools will then use this architecture model, augmented with security information to validate the correct use of the MILS principles as well as generate the security policy of the execution platform, show assurance of security-level isolation, and, ultimately, generate security attack surfaces and trees using the architecture to check the correct implementation of the security policy. Using the architecture to generate security attacks, and thus test the implementation, helps the tool generate attacks tailored to a specific architecture.

We will execute this research agenda over the next two years. Our AADL security annex will be proposed to the SAE AADL Subcommittee while security analysis tools and code generators for the security policy configuration will be integrated in the OSATE AADL tool chain.

Acknowledgments

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

5. REFERENCES

[1] Comprehensive Experimental Analyses of Automotive Attack Surfaces.

[2] J. Alves-Foss, W. S. Harrison, P. Oman, and C. Taylor. The MILS Architecture for High-Assurance Embedded Systems. *International Journal of Embedded Systems*, 2005.

[3] C. Boettcher, R. DeLong, J. Rushby, and W. Sifre. The mils component integration approach to secure information sharing. In *Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th*, pages 1–C. IEEE, 2008.

[4] J. Delange and P. Feiler. Architecture fault modeling with the aadl error-model annex. In *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on*, pages 361–368. IEEE, 2014.

[5] J. Delange and P. Feiler. Incremental latency analysis of heterogeneous cyber-physical systems. In *Third International Workshop on Real-Time and Distributed Computing in Emerging Applications (REACTION)*. Universidad Carlos III de Madrid, 2014.

[6] J. Delange, L. Pautet, and F. Kordon. Design, implementation and verification of MILS systems. *Software: Practice and Experience*, 42(7):799–816, 2012.

[7] J. Hansson, B. Lewis, J. Hugues, L. Wrage, P. H. Feiler, and J. Morley. Model-Based Verification of Security and Non-Functional Behavior using AADL. *IEEE Security and Privacy Magazine*, 2009.

[8] R. Hawkins, T. Kelly, and I. Habli. Developing Assurance Cases for D-MILS Systems. *International Workshop 2015 on MILS: Architecture and Assurance for Secure Systems*.

[9] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[10] P. K. Manadhata and J. M. Wing. An attack surface metric. *Software Engineering, IEEE Transactions on*, 37(3):371–386, 2011.

[11] C. Miller and C. Valasek. Adventures in automotive networks and control units. In *DEF CON 21 Hacking Conference. Las Vegas, NV: DEF CON*. Las Vegas, NV: DEF CON, 2013.

[12] C. Miller and C. Valasek. Remote Exploitation of an Unaltered Passenger Vehicle. In *Black Hat Conference*, 2015.

[13] J. Rushby. The Design and Verification of Secure Systems. In *Eighth ACM Symposium on Operating System Principles (SOSP)*, pages 12–21, Asilomar, CA, Dec. 1981. (ACM *Operating Systems Review*, Vol. 15, No. 5).

[14] SAE International. *AS5506 - Architecture Analysis and Design Language (AADL)*, 2012.

[15] SAE International. *AADL Error Model Annex, (Standards Document AS5506/1, 2006. in revision as Document AS5506/3 2014*, 2014.

[16] B. Schneier. Attack trees. *Dr. Dobb's Journal*, 24(12):21–29, 1999.

[17] K. van der Pol and T. Noll. Security Type Checking for MILS-AADL Specifications. *International Workshop 2015 on MILS: Architecture and Assurance for Secure Systems*.

[18] B. Zalila, J. Hugues, and L. Pautet. *Ocarina user guide*. TELECOM ParisTech.