Formalized Traceability for Security Assurance in AUTOSAR SecOC: A Product Line Engineering Perspective [Under Review]

SAMINA KANWAL*, Vrije Universiteit Amsterdam, The Netherlands

MAURICIO VERANO MERINO*, Vrije Universiteit Amsterdam, The Netherlands

WAN FOKKINK*, Vrije Universiteit Amsterdam, The Netherlands

Customized flexibility is an essential factor in the development of the software-defined vehicle era, the complexity of embedded architectures has reached unprecedented levels. In parallel, the attack surface of modern vehicles has expanded significantly due to vehicle-to-X communication, OTA updates, and increased connectivity within the in-vehicle network domain. The AUTOSAR Secure Onboard Communication (SecOC) module addresses this challenge by ensuring message authenticity and integrity across distributed ECUs. However, integrating SecOC security requirements consistently across a vehicle product line often spanning dozens or hundreds of variants requires a rigorously defined and formally verifiable traceability strategy. This paper introduces a framework of structured requirements traceability rules within a Product Line Engineering (PLE) approach, specifically designed to support security assurance, risk traceability, and compliance in complex, variant-rich automotive environments. Our framework disciplines the linkage between high-level security requirements, such as derived from standard requirements or TARA processes, and concrete architectural features or configurations. By embedding requirements directly into feature and sub-feature hierarchies and conditioning them through logical guards, we ensure that security functions like Message Authentication Code (MAC) verification, freshness checks, and startup authentication are only activated in relevant contexts. Finally, our framework facilitates change impact analysis and modular reuse of security components across variants, critical for reducing lifecycle costs and accelerating feature deployment. In practice, this structured approach transforms security assurance from a reactive task into a proactive, model-driven discipline that aligns technical rigor with real-world engineering efficiency.

Additional Key Words and Phrases: Do, Not, Us, This, Code, Put, the, Correct, Terms, for, Your, Paper

ACM Reference Format:

Samina Kanwal, Mauricio Verano Merino, and Wan Fokkink. 2018. Formalized Traceability for Security Assurance in AUTOSAR SecOC: A Product Line Engineering Perspective [Under Review]. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX).* ACM, New York, NY, USA, 18 pages. https://doi.org/XXXXXXX.XXXXXXX

Authors' Contact Information: Samina Kanwal, s.kanwal@vu.nl, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands; Mauricio Verano Merino, m.verano.merino@vu.nl, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands; Wan Fokkink, w.j.fokkink@vu.nl, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands.

1 Introduction

Regulatory and standardization bodies such as FAA[1], EASA[2], and IEC [3] impose stringent safety requirements through standards such as DO-178C (avionics software)[4], ISO 26262 (automotive functional safety)[5], and IEC 61508 (industrial functional safety)[6]. Similarly, cybersecurity standards are increasingly recognized as essential components of system assurance, as vulnerabilities in software and hardware can directly impact system safety and reliability. In the automotive domain, for instance, the AUTOSAR[7] framework plays a pivotal role by providing a standardized software platform that supports the consistent application of both ISO 26262 and ISO/SAE 21434[8]. These requirements are especially critical in the context of large-scale, complex, and reconfigurable safety-critical systems that integrate diverse subsystems, heterogeneous technologies, and adaptive functionalities [17]. The dynamic and interconnected nature of these systems increases the attack surface of modern vehicles due to vehicle-to-everything (V2X) communication, On-The-Air (OTA) updates, and heightened internal connectivity within the in-vehicle network domain. These connectivity features, while enabling advanced functionality and user convenience, introduce substantial cybersecurity challenges by exposing critical control and data pathways to potential remote and local threats.

Consequently, these standards and frameworks demand structured, auditable justification of design decisions, aligning safety and security objectives with regulatory intent and system-level assurance requirements. However, the provision of these justifications remains a time-consuming and error-prone process, as it requires a process engineer to verify the fulfillment of hundreds of requirements based on the evidence produced by numerous process artifacts and activities [11, 18, 32]. In such contexts, engineering process lines and the systematic identification of commonalities and variabilities are essential for scaling assurance across variable systems. Product Line Engineering (PLE) offers a paradigm for managing reuse and customization, but it is still widely regarded as a challenging, labor-intensive, and costly endeavor. Notably, product lines often evolve from existing variants rather than being developed from scratch [24], which further complicates the assurance landscape. To ensure regulatory compliance across all valid configurations, a detailed analysis of commonalities and variabilities is required. However, variability introduces combinatorial design decisions in which security claims must remain valid, traceable, and auditable under all permissible feature selections. There are some published studies that focused on the mapping regulations and variability management of the process line [12, 13, 15, 29, 31] to facilitate the domain engineer. However, to the best of our knowledge, there have been no systematic efforts to enable compliance checking that rigorously establishes traceable linkages between high-level security requirements, such as those derived from standards (e.g., AUTOSAR SecOC) or threat analysis and risk assessment (TARA) processes [20], and concrete architectural features-based variability or configurations process models.

To address the aforementioned limitations, we propose a framework for the formalization of variability mapping rules, which define how features relate to implementation artifacts and assurance evidence. By making these mappings explicit and machine-verifiable, domain engineer can systematically propagate regulatory requirements through the configuration space and automate consistency checks between selected features, their associated behaviors, and corresponding compliance evidence. This not only reduces manual verification effort but also strengthens the reusability

[1]Federal Aviation Administration (FAA) https://www.faa.gov
[2]European Union Aviation Safety Agency (EASA)https://www.easa.europa.eu
[3]International Electrotechnical Commission (IEC) https://www.iec.ch
[4]RTCA DO-178C: https://www.rtca.org/
[5]ISO 26262 Standard: https://www.iso.org/standard/68383.html
[6]IEC 61508 Standard: https://webstore.iec.ch/publication/5510
[7]AUTOSAR: AUTomotive Open System ARchitecture https://www.autosar.org
[8]ISO/SAE 21434 Standard: https://www.iso.org/standard/70918.html

and completeness of assurance across variants, thereby enhancing the overall integrity of compliance processes in safety- and security-critical domains. Our focus is on the semantic containment boundary of each requirement. That is, it defines (within the feature model) a requirement to be realized. This is critical in avoiding under-implementation (missing requirements in valid configurations) and over-implementation that results in unnecessarily complex or resource-intensive behavior in variant environments where such requirements are not justified. We illustrate our vision by applying it to a small excerpt from AUTOSAR's Secure Onboard Communication (SecOC) [6] module as vehicle architectures grow increasingly complex, spanning dozens of product variants across platforms and markets, ensuring consistent and verifiable implementation of security requirements becomes not just advantageous but essential. By embedding requirements directly into feature and sub-feature hierarchies, and conditioning them through logical guards, we ensure that security functions like Message Authentication Code (MAC) verification, freshness checks, and startup authentication are only activated in relevant contexts, reducing both resource overhead and the risk of misconfiguration. Moreover, bidirectional traceability, combined with typed artifact binding tagged (e.g., CodeModule, ARXML, HeaderFile, TestCase), enables fast, automated compliance validation. Beyond compliance, these rules facilitate meaningful change impact analysis and modular reuse of security components across variants, which is critical for reducing lifecycle costs and accelerating feature deployment. This paper aims to address a compliance-checking approach that links feature selections in SPLs with the secure communication guarantees mandated by SecOC, thus supporting traceability, auditability, and certification readiness in safety- and security-critical automotive systems.

The rest of this paper is organized as follows: Section 2 provides essential background information. Section 3 presents the proposed approach for compliance checking vision, generation of variability and configuration models, and realization. Section **??** demonstrates the effectiveness of the proposed approach for the AUTOSAR SecOC standard. Section 8 discusses the related work. Section 6 presents the limitations and future research directions. Finally, Section 9 concludes the paper with some final remarks.

2 Background

2.1 AUTOSAR SecOC

Modern automotive systems are becoming increasingly complex, interconnected, and software-driven. This evolution, driven by the integration of advanced features such as vehicle-to-everything (V2X) communication, OTA updates, and centralized domain controllers, has significantly expanded the attack surface of vehicles, raising critical concerns for functional safety and cybersecurity. As a result, ensuring compliance with security standards such as ISO/SAE 21434 [2] and AUTOSAR SecOC [6] has become essential for safeguarding in-vehicle communication and maintaining system integrity.

Within the AUTOSAR framework, which provides a standardized software architecture for automotive systems, the SecOC module plays a central role in enabling secure communication between ECUs. SecOC ensures the authenticity and freshness of exchanged PDUs by incorporating cryptographic mechanisms, such as message authentication codes, that are validated at both the sender and receiver ends. This module integrates into the AUTOSAR layered architecture via the PDU Router (PduR), forming a critical part of the secure communication stack. According to the specification [6], both the transmitting and receiving ECUs must implement the SecOC module, and the configuration of authentication parameters must align with the system's security requirements and performance constraints.

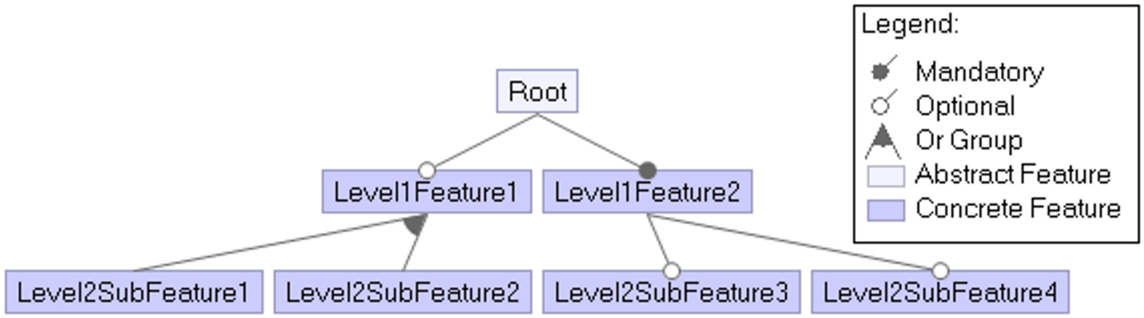

Fig. 1. Feature Modeling Elements

2.2 Compliance Checking

Compliance checking refers to the systematic verification that a system's design, implementation, and artifacts satisfy applicable regulatory requirements and risk mitigation objectives. Unlike conventional verification methods that focus primarily on functional correctness, compliance checking involves a multi-level traceability chain connecting high-level standards and regulatory intent to concrete system-level security requirements, and ultimately to design models, implementation artifacts, test results, and assurance cases [19]. These requirements are typically risk-informed and assigned varying assurance levels based on the criticality of system components and the potential impact of their failure or compromise.

The process of achieving compliance involves the elicitation of relevant requirements, their formalization into checkable criteria, and the establishment of traceability between these criteria and corresponding system artifacts such as specifications, architectural models, code, tests, risk assessments, and validation reports. Historically, the concept of system trustworthiness and compliance can be traced back to early security assurance models such as the Bell-LaPadula model [9], the Orange Book [1], and the Common Criteria [3], which formalized approaches for verifying access control, confidentiality, and system evaluation. However, in modern systems, compliance frameworks are significantly broader, encompassing both static properties (e.g., memory isolation, interface policies) and dynamic properties (e.g., message freshness, fault recovery).

2.3 Product Line Engineering

A feature is a system characteristic that is visible to and meaningful for the end user. In the context of software systems, particularly in SPLs, a feature represents a specific attribute or functionality that encapsulates both commonalities and variabilities across different product variants. However, the variability management introduces significant challenges in ensuring that each valid configuration remains compliant with applicable security standards. Feature interactions may lead to unexpected behaviors or inconsistencies in system behavior [5], which complicates certification processes. A systematic analysis of variability management and traceability across configurations provides a structured approach to detecting and managing potential inconsistencies, enhancing stability and adaptability across product lines.

Generally, a product line is specified using a feature model, in which features are represented as nodes in a tree structure [16]. Four primary parental relationships characterize feature dependencies, as exemplified in Figure 1: (1) Mandatory: if the parent feature is selected, the child feature must also be selected (denoted by a filled circle at the child feature end); (2) Optional: if the parent feature is selected, the child feature may or may not be included (represented by an empty circle at the child end); (3) Alternative (Xor): if the parent feature is included, only one of its children may be selected; and (4) Or: one or more of the child features may be selected when the parent feature is included. Cross-tree constraints, such as Excludes (two features cannot coexist) and Requires (one feature necessitates another), further control feature combinations.

3 Methodology

In this section, we propose a methodology to integrate model-based PL with formal traceability and security assurance. Our method as shown in Figure 2 defines a structured set of formal rules to govern the mapping of cybersecurity requirements to system features and implementation artifacts. This process is inherently iterative, as the domain engineer evolves and regulatory expectations mature over time. Therefore, the core principle of our approach is the systematic identification of variation points and the formal conditions under which specific cybersecurity requirements become active. This ensures that configurable features are consistently and transparently linked to their technical realization, facilitating end-to-end traceability across the reconfigurable systems.

To support formal variability management and traceability, we apply a model-based methodology grounded in a structured Ecore-based metamodel as presented in our previous work [17]. We extended this metamodel to define the core syntax and semantics of feature modelling constructs, which collectively capture the configurable dimensions of the system. It identifies the common and variable features of the product line and represents them as feature elements with associated variant options. Each variation point is captured with formal relationships (e.g., alternative or optional variants), ensuring the product line scope is explicitly modelled. Moreover, we also specified context condition elements to denote contextual circumstances or constraints under which certain variants are applicable. These context conditions formalize environmental or usage scenarios (e.g., operating environment, regulatory constraints, etc.) that impact feature selection. Incorporating context in this way allows the model to capture when specific features or security controls are needed, reflecting the notion that traceability links exist within the context. This ensures that the security engineering concepts are woven into the variability and traceability formalization from the start. The domain engineers identify mechanisms that can: (1) accurately map requirements to specific features or sub-features in the product line model; (2) capture the semantic intent of each requirement, such as its relation to assets, assurance evidence, or verification outcomes etc., ; (3) maintain configuration-aware trace links that are valid across product variants; and (4) integrate with diverse work products, including security models, configuration definitions, and assurance artifacts; (5) ensure complete coverage of all active requirements in every valid product configuration, preserving system compliance and security assurance across the variability space; and (6) enforce bidirectional and policy-constrained traceability semantics, ensuring trace links are only established between permitted element types and that navigability is preserved in both directions.

Throughout the construction of the process model, the traceability information is preserved and embedded, i.e., each element in the process model (each step, component or module) can be annotated in the variability model. For instance, a process step responsible for verifying the freshness of a received PDU within the SecOC_RxIndication() function can be annotated with a trace link to the security requirement “ensure message freshness verification” and further back to the threat “replay attack.” This process step is also linked to the feature Counter-basedFreshness in the variability model, which activates the counter-based freshness strategy in the selected configuration. The trace enables auditors or tools to navigate from the active function at runtime to the originating requirement and the rationale for its inclusion (i.e., mitigation of replay attacks). Therefore, bidirectional traceability is preserved even in the final product artifact: one

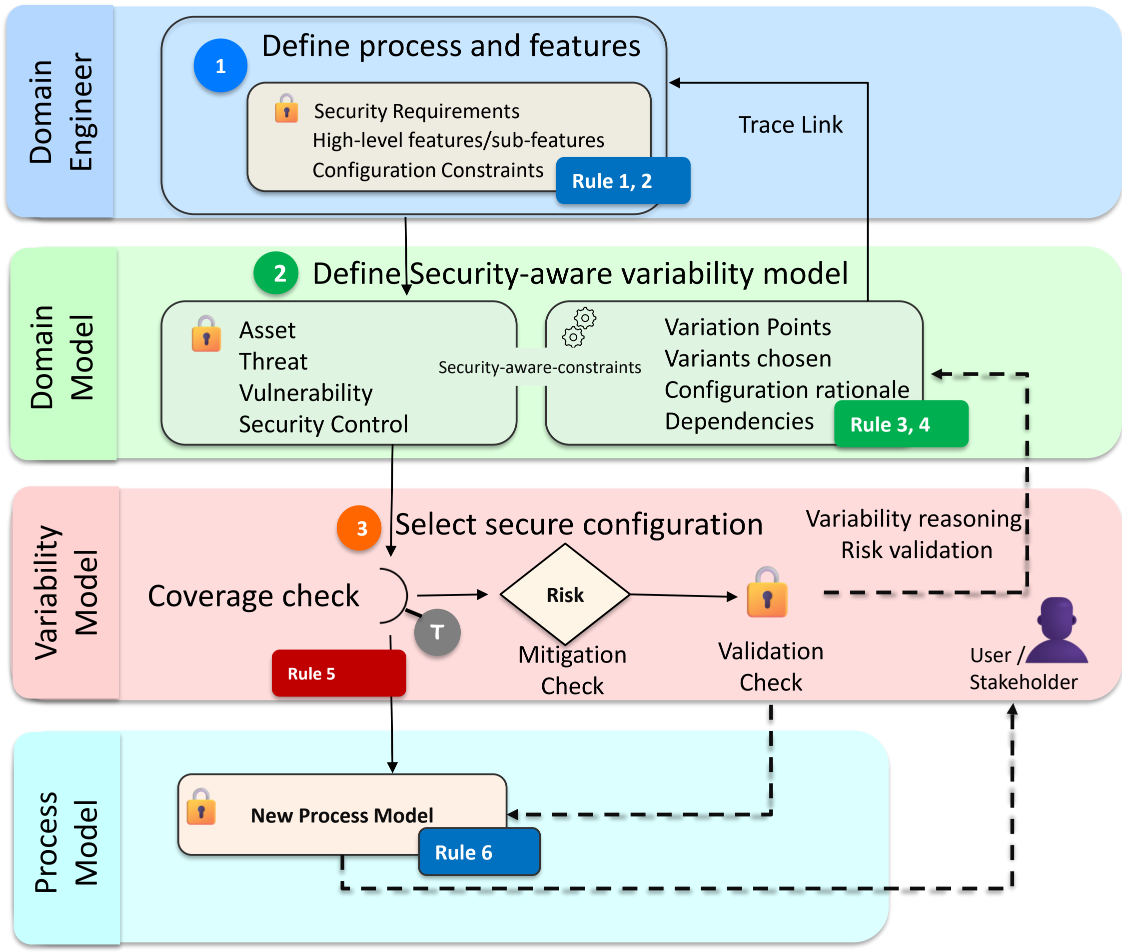

Fig. 2. Overview of the Proposed Approach for Process Variability Management

can trace from any element in the process model back to the requirements and features in the product line model, and from any requirement to the implemented elements in the process model. The benefit of this is twofold: (1) Verification – it is easy to verify that every security requirement has been implemented in the process (no requirement was lost in the transition from model to product), and (2) Impact analysis – if the process model is to be changed or if a new threat emerges, engineers can quickly identify which products and which parts of those products are affected (since the links tie back into the variability model and possibly to other products sharing that feature).

4 Metamodel for Security Process Variability Management

In this section, we present the full integration of PLE with security assurance and traceability as shown in Figure 3. The proposed metamodel establishes a robust foundation for building secure software product lines. It enables domain experts to perform security trade-off analyses during feature selection and to maintain compliance by supporting comprehensive threat modeling, risk assessment, and traceability of security requirements across the product lines. This

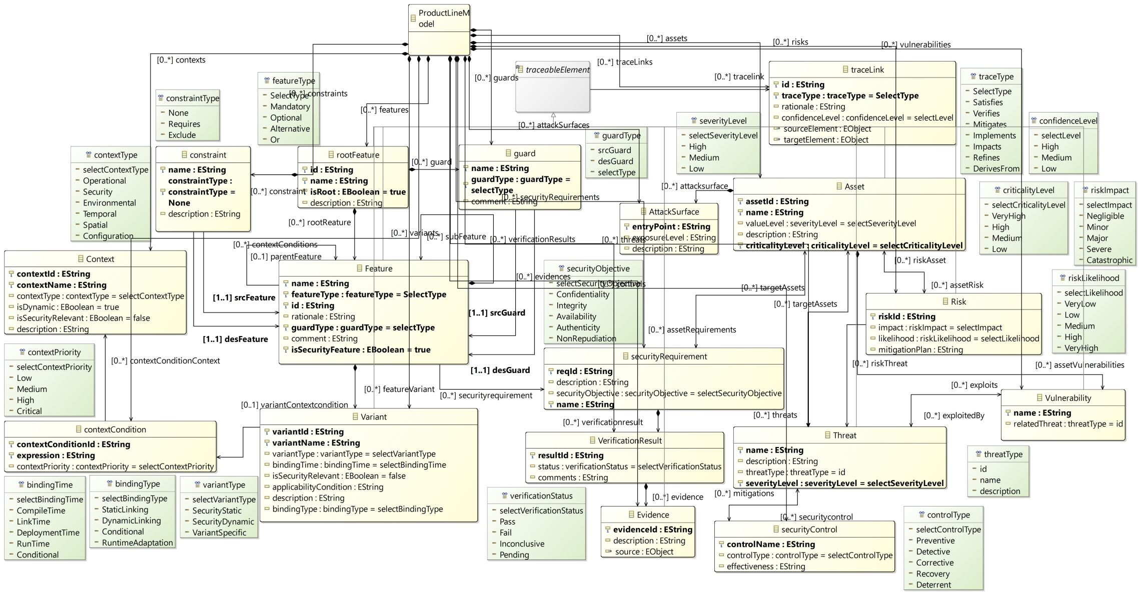

Fig. 3. Proposed metamodel for Security Process Variability Management

means that the model facilitates bidirectional navigation, allowing domain experts to address critical security-related questions such as: "If this feature is selected, what associated risks and security requirements must be considered?", "Which product variants are affected by a given threat scenario?", or "What controls are in place to mitigate a specific vulnerability across the product line?" Therefore, each relationship decision was made with the aim of modeling traceable security requirements and risk mitigations in a PLE context, ensuring that security considerations are not siloed but are intertwined with feature modeling and product configuration. The metamodel supports parent–child feature hierarchies, rich associations between PLE core elements and security artifacts, and bidirectional navigability for traceability.

4.0.1 Product Line Feature Model (PLE Structure) with Variant and Context Integration Associations. In the proposed metamodel, the *ProductLineModel* element serves as the root container, capturing both product line design and security aspects in a unified structure. It includes key components such as the feature model (linked via *rootFeature*) and security-related elements (security requirements, assets, threats, etc.). In feature modeling, the root feature (a subclass of *traceableElement*) is the abstract representation of the product line itself, with all other features organized in a tree hierarchy [22]. By subclassing *traceableElement*, the root feature and its descendants can participate in traceability links (e.g., linking features to requirements or other artifacts). The *Feature* class, a subclass of *traceableElement*, represents an individual feature within the feature model and is uniquely identified by a name and identifier. Each feature includes a *featureType* attribute, defined as an enumeration capturing the feature's variability semantics. Supported types include: *Mandatory*, indicating the feature is required when its parent is selected; *Optional*, where the feature may or may not be included with its parent; *Alternative*, representing exclusive selection among sibling features (XOR); Or, allowing one or more features within a group to be selected. Similarly, to prevent invalid combinations and allow the modeler to encode domain knowledge about feature compatibility, we introduced constraints and guard classes to ensure consistency of feature selections across the product line. However, the model provides a notion of Variant and Context to handle concrete product instances or dynamic variation points, enabling multi-dimensional variability [28].

In addition to individual features, entire product variants (configurations) and their usage context must be connected to security concerns. The Variant class gains a reference (selectedFeatures: Feature [0..*]) to denote which features are included in that product variant. This is a fundamental PLE relationship where each product is defined by a set of feature selections. We choose composition because a Variant is not a globally reusable concept; it exists in the context of a specific feature or variation point. For instance, a feature representing a generic security function can include variant instances that denote concrete implementation choices, distinguished through an attribute such as variantType (e.g., SecurityStatic, SecurityDynamic). However, to support context-aware variability, the Variant class is associated with the ContextCondition class, which specifies the applicability constraints under which a variant is selected [21]. This relationship is modeled as a non-containment reference (variantContextCondition: ContextCondition [0..*]), indicating that a variant can be linked to a condition that must be true for it to be valid or selected. This design enables the specification of rules such as "Variant X is only applicable in operational context with high priority" or more complex boolean expressions over contextual attributes. These associations create a feature model structure that combines a tree (using compositions for root–features–variants) hierarchy with selectively introduced cross-references. This hierarchical containment ensures the fundamental feature selections are organized in a well-defined structure. Accordingly, the cross-cutting references (requires/excludes, variant-to-context conditions) provide additional flexibility, allowing for the modeling of product line constraints and dynamic variations without breaking the tree. This dual approach thus accommodates both requirements: a strict hierarchical feature model as the structural backbone, and loosely coupled references to encode variability and dependency rules.

4.0.2 Security Assurance and Traceability Associations. A standout aspect of this meta-model is the detailed inclusion of security engineering concepts, aligned with standard risk management [2, 7, 10]. However, to promote modularity and reduce coupling between security and operational concerns, we only associated *SecurityRequirement* class with *Feature* rather than directly with domain-specific concepts such as Asset, Threat, or Risk. This separation enables independent evolution of the security and threat/risk models [4, 27]. Beyond linking PLE classes to security classes, we also refine relationships among the security classes themselves to support end-to-end traceability of requirements and mitigations. The goal is to model the chain from identified threats or vulnerabilities through to the requirements and controls that address them, all within the context of the product line.

Since security requirements are often defined in relation to the assets they are intended to protect, the metamodel establishes a bidirectional association between the *SecurityRequirement* and *Asset* classes. Specifically, a security requirement may reference one or more target assets via *SecurityRequirement.targetAssets [0..*]*, with the inverse captured by *assetRequirements [0..*]*. This relationship allows requirements to explicitly state which assets they safeguard, while also enabling each asset to list the security requirements applicable to it. Such bidirectional traceability supports both asset-focused and requirement-focused analyses, facilitating the assessment of protection coverage across the product line. Similarly, we created a bidirectional non-containment reference between *Asset* and *Threat* classes. For example, an asset may be associated with multiple threats via *Asset.threats [0..*]*, while each threat specifies its intended targets through *threatTargetAssets [0..*]*. This structure reflects the fact that threats target specific assets, and assets, in turn, may be exposed to multiple threats. Establishing this explicit linkage supports threat-centric analyses, e.g., identifying which assets are vulnerable, and asset-centric evaluations by determining the threat landscape for a given asset. We use a plain reference (not composition), considering that the threat is not a part of the asset lifecycle, but an external danger. Since a vulnerability is essentially a weakness of an asset, we use a composition (containment reference) from Asset to Vulnerability (e.g., assetVulnerabilities: Vulnerability[0..*]) that is contained within the Asset. This relationship ensures that if an Asset is removed, the corresponding vulnerabilities are removed too, reflecting that vulnerabilities are only relevant in the context of that asset. Furthermore, a threat may reference one or more vulnerabilities (exploits: Vulnerability[0..*]), while a vulnerability may be associated with one or more threats (exploitedBy: Threat[0..*]). To capture this bidirectional association, we introduced a non-containment reference between Threat and Vulnerability, defined as opposites. This modeling decision emphasizes that Threats and Vulnerabilities are independent entities, while still enabling a flexible many-to-many relationship that supports loose coupling between them.

According to security principles [8, 26], controls are employed to reduce either the likelihood or the impact of threats exploiting existing vulnerabilities. To represent this relationship, we defined a bidirectional reference between Threat and SecurityControl by specifying e-opposites on both classes. Concretely, a threat maintains a reference to the set of controls that mitigate it (mitigations: SecurityControl[0..*]), while a SecurityControl maintains a reference to the threats it mitigates (mitigatesThreats: Threat[0..*]). The Risk class represents a security risk scenario that arises when a threat has the potential to exploit a vulnerability on a given asset, thereby causing an adverse impact. To capture this, we introduce explicit references (riskThreat: Threat [0..*]), (riskVulnerability: Vulnerability [0..*]) (opposite vulnerabilityRisk) and (riskAsset: Asset [0..*]) (optional; potentially redundant when the referenced vulnerabilityAsset is defined). These associations ensure navigability and maintain a loose coupling between entities while allowing a Risk instance to specify precisely what the harmful action is, how it becomes feasible, and which asset is affected. For example, in a SecOC scenario [6], a risk can be linked to the threat "ReplayAttack" to the vulnerability "MissingFreshnessValidation" and optionally to the asset "SecuredIComSignal" of a specific feature. This formalization is consistent with the security objectives of SecOC, which focus on authenticity and freshness of inter-ECU communication, and allows both cases where risks involve concrete vulnerabilities (e.g., improper freshness counter handling) and cases where they are threat-driven without a discrete flaw (e.g., insider misuse). The optional reference to the asset is provided for disambiguation, although in many cases the vulnerability already implies the affected asset. Furthermore, the AttackSurface class is defined as a compositional element of the Asset class, reflecting the principle that entry points and exposure points are meaningful only in the context of the asset to which they belong. Accordingly, an asset can contain zero or more AttackSurface instances (e.g., assetAttackSurfaces: attackSurface[0..*] as a containment reference). This composition ensures that descriptions of attack surfaces are owned by the corresponding asset and do not exist independently, thereby preserving semantic consistency between assets and their possible points of exposure. Moreover, attributes of the AttackSurface class, such as entryPoint and exposureLevel, characterize specific facets of the asset's exposure, enabling how the asset may be approached, accessed, or exploited. These associations ensure that each class instance remains an independent entity, with no inheritance relationships imposed between them, while the defined references provide the logical connections necessary for modeling their interactions. To support systematic traceability across the entire model, an abstract superclass TraceableElement is introduced to enable consistent referencing and trace management. However, to preserve the flexibility and decoupling within the metamodel, the traceLink class is then defined with reference source: TraceableElement and target: TraceableElement each pointing to a generic EObject that can connect any two artifacts in case a certain relationship is needed that is not covered by a dedicated association. For example, to trace a relationship "Feature implements SecurityRequirement", we can create a traceLink where sourceElement is the Feature, targetElement is the SecurityRequirement, and set traceType = Implements. Similarly, another trace might link a securityRequirement to a Threat with traceType = Mitigates (meaning that requirement mitigates that threat), or link an Evidence to a securityRequirement with traceType = Verifies (evidence verifies the requirement). The enumerated traceType values (Satisfies, Verifies, Mitigates, Implements, Impacts, Refines, DerivesFrom) defines the semantics specification of the traceLink instance.

In conclusion, we choose the above associations to balance strict hierarchy vs. flexibility: compositions are used for natural containment hierarchies (feature trees, security parts) and references for cross-cutting links (constraints, mappings, trace links). Each decision is made with both product-line engineering and security verification in mind, which allows organizations to perform security trade-off analyses, maintain compliance by tracing security requirement through to implementation in feature, and adapt to different context on a single, traceable model, making sure that security is fully built-in to the product line, not bolted on.

5 Evaluation

5.1 Formalization of Product Line Engineering, Requirements Artefacts, and Traceability

In the literature, traceability research has predominantly focused on point-to-point mappings between requirements and system artifacts [14, 23, 25, 30]. However, SPLE introduces additional dimensions, such as feature hierarchies, guarded configurations, and variant-specific applicability that demand more nuanced mapping strategies. In this context, traceability must support not only the presence of requirement implementation but also its semantic, structural, and configurational alignment across a family of products. Specifically, PLE demands traceability mechanisms that can: (1) accurately map requirements to specific features or sub-features in the product line model; (2) capture the semantic intent of each requirement, such as its relation to assets, assurance evidence, or verification outcomes etc., ; (3) maintain configuration-aware trace links that are valid across product variants; and (4) integrate with diverse work products, including security models, configuration definitions, and assurance artifacts.

To address these challenges, we propose a set of prescriptive mapping rules that define how standards-based requirements should be systematically related to elements of product line models and configuration artifacts. These rules are rooted in principles from requirement engineering, PLE and standards-based systems engineering (e.g., ISO 26262, SAE 21434, ASPICE). Each rule is associated with a distinct intent. For example, promoting granularity (Rule 1), supporting standard compliance (Rule 3), or enabling variant completeness checks (Rule 7) can be operationalized using modern toolchains, including pure::variants, FeatureIDE, Capra, and IBM ELM.

Rule 1: Map Requirement to Feature/Sub-feature Hierarchically

Intent: Every requirement must map to the lowest applicable level in the feature hierarchy for granularity.

If a requirement R_i expresses a system or software capability, then there must exist a feature F_j in the PLE feature model such that:

$$R_i \rightarrow F_j, \quad \text{where } F_j \in \{\text{Feature} \cup \text{SubFeature}\}$$
$$\text{and } \forall F_k \subset F_j : R_i \notin F_k \quad \text{(no more precise mapping exists).}$$

Figure 4 exemplifies the rule of mapping each requirement to the lowest applicable feature level using three representative requirements (R1–R3). R1. “The module shall support freshness counters as a strategy” refers to a specific mechanism within the abstract feature FreshnessStrategySelection, which includes Counter-basedFreshness, Timestamp-basedFreshness, and Gateway-synchronizedFreshness. Since the requirement explicitly mentions counters, it is mapped to Counter-basedFreshness; mapping it to the parent would incorrectly generalize its scope. R2. “The module shall support

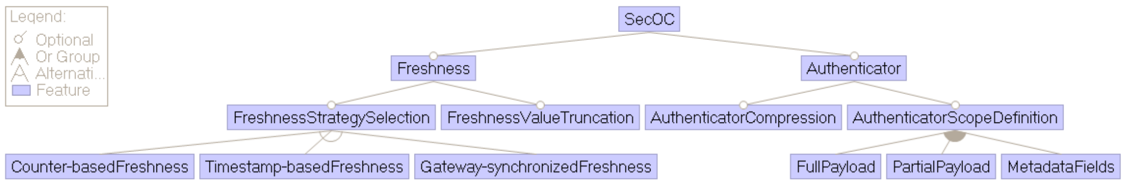

Fig. 4. Requirement-to-feature mapping at the most granular level

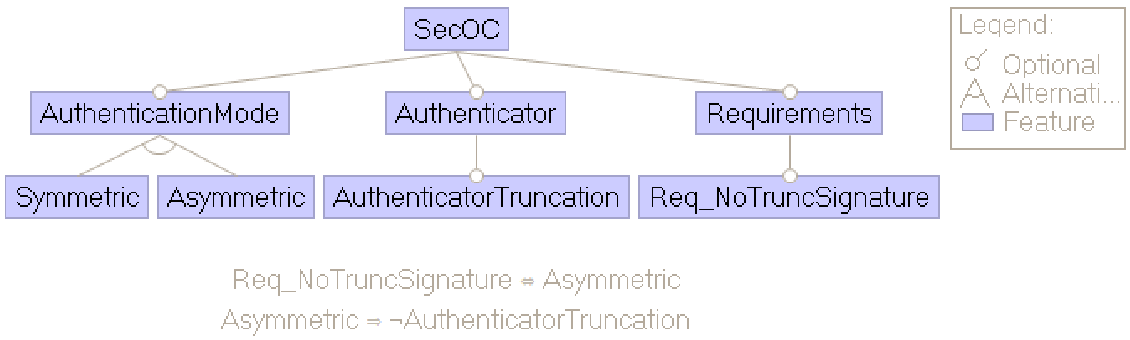

Fig. 5. Constraints via Feature Guards

truncation of the freshness value" specifies an operation rather than a general freshness function. While Freshness captures overall freshness handling, only the sub-feature FreshnessValueTruncation precisely represents truncation. Therefore, R2 is mapped to this sub-feature as the most specific applicable level. R3. "The authentication shall cover the entire payload" defines the coverage constraint of the authenticator. The grouping feature AuthenticatorScopeDefinition includes FullPayload, PartialPayload, and MetadataFields. Because the requirement specifies complete coverage, it maps to FullPayload; mapping it to the parent would introduce ambiguity.

Rule 2: Model Configuration Constraints via Feature Guards

Intent: Express requirement applicability conditions using logical constraints over features.

If a requirement R_i is applicable only under configuration C_k, define a variation point V_k with guard condition G_k, and bind R_i to V_k via:

$$(G_k \implies R_i \text{ active}) \land (\neg G_k \implies R_i \text{ ignored})$$

For example, the variation-point binding is defined as Req_NoTruncSignature ⇔ Asymmetric, indicating that the requirement *Req_NoTruncSignature* becomes active only when the feature *Asymmetric* is selected (G_1 = true) and is ignored otherwise ($\neg G_1$ = true), thus satisfying $(G_1 \implies R_1 active) \land (\neg G_1 \implies R_1 ignored)$. Furthermore, the domain integrity constraint Asymmetric ⇒ ¬ AuthenticatorTruncation ensures that when asymmetric authentication is applied, truncation of the authenticator is disabled, since digital signatures cannot be truncated. Collectively, these rules establish a consistent configuration logic in which the variation point (*AuthenticationMode* → *Asymmetric*) governs the applicability of the requirement and preserves semantic integrity across valid product configurations as shown in Figure 4.

Rule 3: Requirement-to-Security Attribute

Intent: Requirements must be linked to the asset, i.e., actual software artifacts or configurations that implement or realize them. This ensures that each requirement has a tangible, verifiable realization in the system.

For each requirement R_i, define a link to an implementation asset A_j

$$R_i \longrightarrow A_j$$

where

$$A_j \in \{\text{Software Component},\ \text{Module},\ \text{Configuration Parameter},\ \text{Source File},\ \text{Interface},\ \text{Function},\ \text{Calibration Value}\}$$

Similarly, this rule can be extended to describe how `security threats` and `vulnerabilities` are connected to specific `assets`, ensuring that corresponding `mitigation links` can later be traced systematically throughout the model.

$$A_j \longrightarrow T_k \quad \text{or} \quad A_j \longrightarrow V_m$$

where

- A_j: Asset (e.g., Software Component, Configuration Parameter, Interface, Source File)
- T_k: Threat that targets or exploits this asset
- V_m: Vulnerability within or related to this asset

The example shown in Figure 6 represents a configuration asset (`SecOCAuthInfoTxLength`) that controls the length of the authentication field. If this asset is `misconfigured`, it can introduce the vulnerability `V_ImproperLengthConfig`, indicating an insecure parameter setting. This vulnerability enables the threat `T_WeakAuthenticator`, where an attacker may exploit the short authenticator length to forge or replay messages. Thus, the model expresses a clear causal chain: the asset exposes a vulnerability, and the vulnerability leads to a threat, forming a logical trace that supports security analysis, which can also be extended to mitigation planning. Therefore, the mapping rule covers the full security analysis as follows.

$$\text{Requirement} \longrightarrow \text{Asset} \longrightarrow \text{Vulnerability} \longrightarrow \text{Threat}$$

Rule 4: Require Coverage Check for All Configured Variants

Intent: Ensure compliance across all valid product configurations.

For each valid configuration C_x, the active features $F(C_x)$ must collectively satisfy all active requirements $R(C_x)$ such that:

$$\forall C_x : \text{Coverage}(F(C_x), R(C_x)) = 100\%$$

In this example shown in Figure 7, the rule is demonstrated through a feature model that ensures every valid product configuration satisfies all active requirements. The model defines two requirements, `Req_NoTruncSignature` and `Req_CounterFreshness`, each bound to specific feature conditions through bidirectional constraints (<->). The

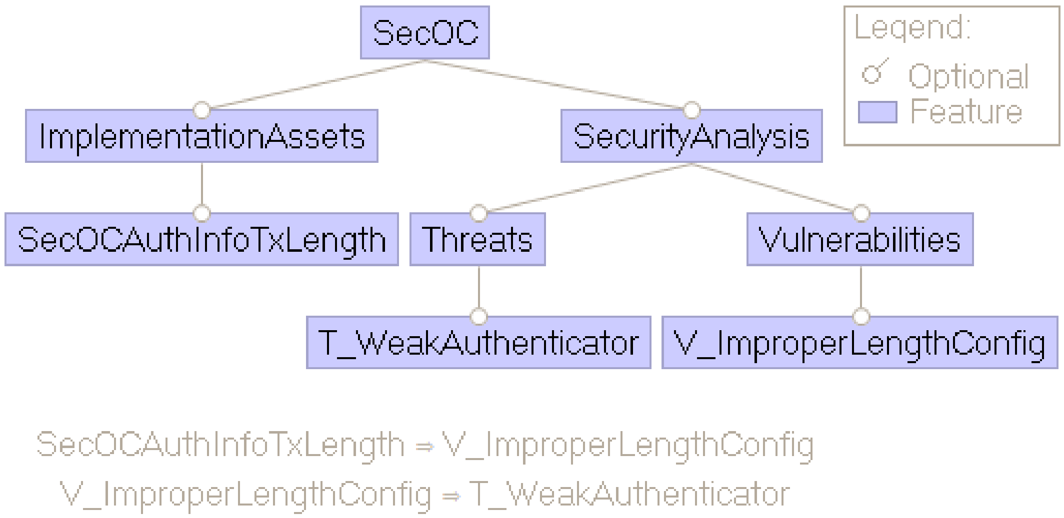

Fig. 6. Requirement-to-Asset-to-Threat/Vulnerability Relation

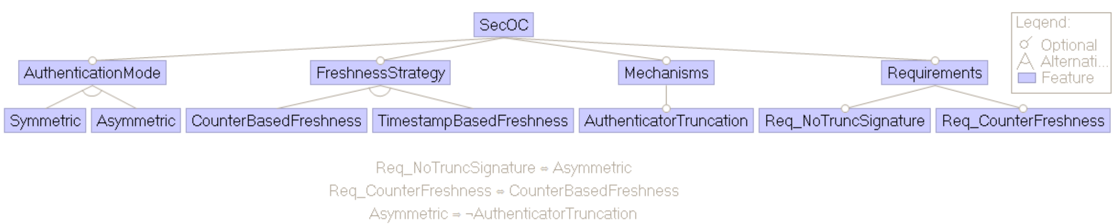

Fig. 7. Requirement coverage check for all configured variants

first constraint, `Req_NoTruncSignature <-> Asymmetric`, ensures that the requirement to prevent signature truncation is only active when the `Asymmetric` authentication mode is selected. Similarly, `Req_CounterFreshness <-> CounterBasedFreshness` activates the freshness-related requirement only when the `CounterBasedFreshness` strategy is configured. An additional constraint, `Asymmetric -> !AuthenticatorTruncation`, enforces that when the `Asymmetric` mode is active, truncation is explicitly disabled, thereby satisfying the associated requirement. Collectively, these logical relationships guarantee that for every valid configuration C_x, the active feature set $F(C_x)$ satisfies all corresponding active requirements $R(C_x)$.

Rule 5: Requirements Classification for SPL Binding

Intent: Classify requirements based on whether they are invariant or variant across product lines.

For each R_i:

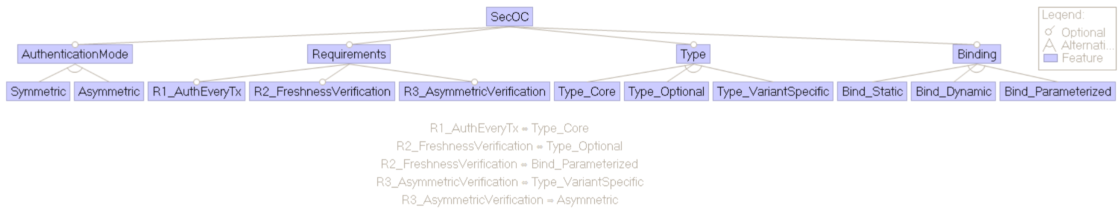

Fig. 8. Classification of requirements into type and binding categories

$$\text{Type}(R_i) \in \{\texttt{Core}, \texttt{Optional}, \texttt{Variant-specific}\}$$

$$\text{Binding}(R_i) \in \{\texttt{Static}, \texttt{Dynamic}, \texttt{Parameterized}\}$$

This rule demonstrates, as shown in Figure 8, how requirements in a SPLE can be systematically classified and linked to their binding characteristics. Each requirement is assigned a type `Core`, `Optional`, or `Variant-specific` depending on whether it applies universally, conditionally, or only to specific product variants. Similarly, each requirement is associated with a binding category `Static`, `Dynamic`, or `Parameterized`, which indicates when it becomes fixed within the product lifecycle. For example, a core requirement such as `R1_AuthEveryTx` is statically bound because it is invariant across all configurations, while an optional requirement like `R2_FreshnessVerification` depends on configuration parameters and is therefore parameterized. A variant-specific requirement such as `R3_AsymmetricVerification` applies only when the `Asymmetric` feature is active and is dynamically bound at runtime. Hence, these classifications ensure consistent requirement management in variable product-line configurations.

Rule 6: Bidirectional Trace Must Exist

Intent: A trace link between any two traceable elements a and b is valid only if the tuple of their metatypes and trace type is permitted by the policy relation Allow. For every existing link, an inverse must exist as a derived navigation, ensuring semantic bidirectionality. This rule guarantees that traceability relationships remain logically sound, navigable in both directions, and adaptable to any artifact combination defined within the SPL.

To define the bidirectional traceability in SPL, our framework specifies the following;

Universes and Typing

$$\tau : U \longrightarrow T$$

Where

- U be the universe of all artifacts (generic `traceable elements`)
- T be the set of artifacts (types), e.g.,

$$T = \{\texttt{Requirement}, \texttt{Feature}, \texttt{Threat}, \texttt{Vulnerability}, \texttt{Evidence}, \ldots\}.$$

Trace types with inverses

Let TT be the set of trace types (labels), e.g.,

$$TT \supseteq \{\texttt{Implements}, \texttt{ImplementedBy}, \texttt{Mitigates}, \texttt{MitigatedBy}, \texttt{Verifies}, \texttt{VerifiedBy}, \ldots\}.$$

- $inv(inv(t)) = t$ (involution property)
- For symmetric trace types, $inv(t) = t$ (e.g., `VerifiedBy`)

Customizability Policy (What is Allowed)

A policy predicate (or relation) controls which triples are permissible:

$$Allow \subseteq T \times T \times TT$$

For fully flexible (default):

$$Allow = T \times T \times TT$$

This represents complete flexibility, meaning any trace type may link any two artifact types.

Trace Links as a Ternary Relation

The (extensional) set of links is defined as a ternary relation:

$$TL \subseteq U \times U \times TT$$

Well-formedness with respect to the policy:

$$\forall(a, b, t) \in TL : (\tau(a), \tau(b), t) \in Allow$$

This ensures that every trace link between two artifacts a and b with trace type t is permitted by the defined policy relation.

Bidirectionality (Semantic Must-Exist Inverse)

This condition implies that for every trace link, an inverse relation must exist as a derived (logical) opposite, ensuring bidirectional navigability without redundant storage.

$$\forall(a, b, t) \in TL : (b, a, inv(t)) \in TL \ \lor \ DerivedInverse(b, a, inv(t))$$

After establishing the requirement for bidirectional consistency, we now formalize the notion of trace-link validity. A trace link is considered valid if and only if the types of its participating artifacts and associated trace type conform to the combinations permitted by the policy. This condition is formally defined as:

$$\forall(a, b, t) \in TL : (a, b, t) \text{ is valid} \Leftrightarrow (\tau(a), \tau(b), t) \in Allow$$
$$\exists(b, a, inv(t)) \in TL \ \lor \ DerivedInverse(b, a, inv(t))$$

This transparency eliminates "blind spots" in the trace network, which often serve as latent points of failure or unmonitored dependencies that can expand the system's attack surface. When each link (a, b, t) must have a corresponding inverse $(b, a, inv(t))$, it becomes impossible for hidden, one-way dependencies to exist undetected; every connection is accountable, navigable, and verifiable in both directions. However, from the security perspective, this ensures that each requirement, mitigation, or verification artifact has a clear and traceable effect across the system configuration. For instance, in an AUTOSAR SecOC context, a requirement that mitigates a cryptographic threat will be reciprocally traceable back from the feature or configuration implementing that control. This enables bidirectional security audits, where both proactive and reactive analyses can be conducted. Therefore, security assurance can be started from a requirement and verify all enforcing components, or start from an implementation and confirm which requirement it satisfies or mitigates. Such completeness reduces the attack surface introduced by unlinked or inconsistently traced assets.

In terms of redundancy reduction, the rule leverages a single, generic trace link structure rather than multiple type-specific trace associations (e.g., RequirementToFeatureLink, FeatureToConfigLink, etc.). This unified model eliminates the need to duplicate trace definitions across model. Henceforth, minimizing the attack surface with fewer classes, fewer specialized relationships, and fewer potential misuse or misconfiguration points. Moreover, since the trace directionality is enforced logically rather than through duplicated inverse relationships, the same semantic bidirectionality is achieved without redundant storage of reverse links, further minimizing model complexity and the risk of inconsistent trace states.

6 Discussion

Formalizing the compliance process within the context of SPL, particularly for security assurance, offers substantial benefits for practitioners and researchers. Specifically, it provides a structured framework to analyse how requirements, features, and other artifacts (such as assets and threats) are interrelated. In a complex domain like automotive security (e.g., AUTOSAR SecOC), such formalization enables researchers and developers to demonstrate to certifiers that all security requirements are correctly implemented and verified. In practice, having requirements tied to variation points (with guard conditions in the model) helps teams ensure that optional or variant-specific requirements are only active when relevant, hence preventing misconfiguration.

To the best of our knowledge, the approach proposed in Figure 2 constitutes the first formalization of bidirectional traceability for security requirements in the context of SPLs and the AUTOSAR-based automotive domain. Unlike previous works that address either requirement, artifact traceability or SPL variability in isolation, our approach unifies both dimensions within a single, formally defined traceability model. It systematically integrates the notions of artifact universes, typed trace relations, admissibility policies, and inverse trace semantics, thereby providing a mathematically precise and tool-enforceable foundation for security-oriented traceability management.

The proposed framework advances the state of the art in several aspects. First, it enables automatic and context-aware trace generation and validation, where trace admissibility is determined by the type and configuration context of the involved artifacts. Second, it supports variant-aware reasoning, ensuring that trace completeness and consistency are verified across all valid configurations of the SPL. Third, and most important, it embeds security requirements from the outset of the engineering process: by incorporating security elements, such as assets with corresponding risks, vulnerabilities, threats, and mitigations, etc., into the same typed universe as functional and architectural artifacts, the model ensures unified, analyzable, and lifecycle consistent security traceability throughout the AUTOSAR development process. Furthermore, the proposed formalization eliminates the need for manual matrix-based linking by allowing automated derivation and bidirectional closure of trace links through formally defined inverse relations and well-formedness constraints. This combination of formal rigor, automation, and context sensitivity makes the approach both theoretically grounded and practically applicable to large-scale, safety- and security-critical product lines.

Despite the advantages, the proposed approach has certain limitations and can introduce biases. One limitation is the overhead and complexity: building and maintaining a detailed feature model with all requirement mappings, constraints, and traceability demands considerable analytical reasoning and domain expertise. For example, a high level requirement like "the ECU shall prevent replay attacks" appears straightforward in natural language but becomes substantially complex when mapped to specific features in a variability-rich model. According to the AUTOSAR

SecOC standard, replay protection can be realized through diverse subfeatures such as ReplayAttackDetectionThreshold, Counter-basedFreshness, or FreshnessValueEncryption, each contributing partially or conditionally to the requirement's compliance. Moreover, the applicability of such features varies based on selected configurations, such as authentication mode (symmetric vs. asymmetric) or security profile variants, introducing the need for guarded constraints and conditional trace links. The requirement can (optionally) be tied to implementation parameters outside the feature model, such as SecOCAuthInfoTxFreshnessValue, requiring cross-model traceability. This interplay between abstract goals, architectural variability, configuration conditions, and runtime assets [9] illustrates why formalizing security requirements is painstaking: as it requires precise disambiguation, multiple-layer mappings, and reasoning under variability. Another limitation is the tool support; most of the development environments lack a native mechanism for defining customised traceability, such as coverage checks. Consequently, without automation, human error could creep in when writing the constraints or establishing the traces. In terms of bias, the proposed approach reflects the bias of the creators. The way requirements are modelled (core vs optional, static vs dynamic binding) or the way trace links are defined could emphasize certain aspects and ignore others. For example, the generic trace link approach (using a flexible TraceLink class with arbitrary source and target) is powerful, but it relies on the user to define the trace type and endpoints correctly. This flexibility could introduce bias in what relationships are deemed important; one engineer might link a requirement to a threat as "mitigates", another might not, leading to inconsistency. In summary, while the formal feature modeling and traceability strategy greatly aids both research and practice by bringing rigor and clarity, one must be mindful of the limitations. However, the benefits of improved coverage, consistency, and insight generally outweigh these challenges for complex security-critical product lines.

7 Threats to Validity

8 Related work

9 Conclusion

References

[1] 1985. Department of Defense Trusted Computer System Evaluation Criteria (DoD Orange Book). https://csrc.nist.gov/publications/detail/dod/1985/12/26/dod-5200-28-std/final.

[2] 2021. ISO/SAE 21434:2021 Road vehicles — Cybersecurity engineering. https://www.iso.org/standard/70918.html.

[3] Norita Mohd Aizuddin. 2001. Common Criteria for Information Technology Security Evaluation (ISO/IEC 15408). https://www.commoncriteriaportal.org/cc/. Accessed 2024.

[4] Nicolas Anquetil, Uirâ Kulesza, Ralf Mitschke, Ana Moreira, Jean-Claude Royer, Andreas Rummler, and André Sousa. 2010. A Model-Driven Traceability Framework for Software Product Lines. *Software Systems Modeling* 9, 1 (2010), 73–96. doi:10.1007/s10270-009-0120-9

[5] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines Concepts and Implementation.* Springer, Springer Heidelberg New York Dordrecht London. doi:10.1007/978-3-642-37521-7

[6] AUTOSAR. 2021. AUTOSAR Specification of Secure Onboard Communication. https://www.autosar.org/fileadmin/user_upload/AUTOSAR_SWS_SecureOnboardCommunication.pdf. Version 4.4.0, Release 21-11.

[7] AUTOSAR. 2023. AUTOSAR SecOC Specification, Release R23-11. Available at: https://www.autosar.org/standards/.

[8] José Victor Barraza de la Paz, Luis Alberto Rodríguez-Picón, Víctor Morales-Rocha, and Silvia V. Torres-Argüelles. 2023. A Systematic Review of Risk Management Methodologies for Complex Organizations in Industry 4.0 and 5.0. *Systems* 11, 5 (2023), 218. doi:10.3390/systems11050218

[9] David Elliott Bell and Leonard J. LaPadula. 1973. Secure computer systems: Mathematical foundations. (1973).

[10] Christian Biermann, Richard May, and Thomas Leich. 2025. Integrating Security into the Product-Line-Engineering Framework: A Security-Engineering Extension. In *Proceedings of the 20th International Conference on Software Technologies - Volume 1: ICSOFT*. INSTICC, SciTePress, 75–86. doi:10.5220/0013489500003964

[11] Julieth Patricia Castellanos Ardila, Barbara Gallina, and Faiz Ul Muram. 2018. Enabling Compliance Checking Against Safety Standards from SPEM 2.0 Process Models. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 45–49. doi:10.1109/SEAA.2018.00017

[9] A runtime asset is any configurable, executable, or stateful element that contributes to the actual behavior of the system during operation and is essential for closing the traceability loop from requirements through design to real-world enforcement.

[12] Chin Pang Cheng, Gloria T. Lau, and Kincho H. Law. 2007. Mapping regulations to industry-specific taxonomies. In *Proceedings of the 11th International Conference on Artificial Intelligence and Law* (Stanford, California) *(ICAIL '07)*. Association for Computing Machinery, New York, NY, USA, 59–63. doi:10.1145/1276318.1276329

[13] Stefan Fenz, Thomas Pruckner, and Arman Manutscheri. 2009. Ontological Mapping of Information Security Best-Practice Guidelines. In *Business Information Systems*, Witold Abramowicz (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 49–60.

[14] Orlena C. Z. Gotel and Anthony C. W. Finkelstein. 1994. An analysis of the requirements traceability problem. In *Proceedings of IEEE International Conference on Requirements Engineering (ICRE)*. IEEE, 94–101. https://api.semanticscholar.org/CorpusID:5870868

[15] Muhammad Atif Javed and Barbara Gallina. 2018. Safety-oriented process line engineering via seamless integration between EPF composer and BVR tool. In *22nd International Systems and Software Product Line Conference (SPLC), Volume 2*. 23–28. doi:10.1145/3236405.3236406

[16] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report. Carnegie-Mellon University Software Engineering Institute.

[17] Samina Kanwal and Wan Fokkink. 2025. A Framework for Model-Based Specification and Verification in Feature-Oriented Software Product Lines. In *Fundamentals of Software Engineering*, Hossein Hojjat and Georgiana Caltais (Eds.). Springer Nature Switzerland, Cham, 61–79.

[18] Samina Kanwal, Faiz Ul Muram, and Muhammad Atif Javed. 2024. Systematic review on contract-based safety assurance and guidance for future research. *Journal of Systems Architecture* 146 (2024), 103036. doi:10.1016/j.sysarc.2023.103036

[19] Gernot Macher, Elisabeth Armbrust, Christian Kreiner, and Dietmar Winkler. 2016. A model-based approach for integrated safety and security assessments. In *Proceedings of the 2nd International Workshop on Safety and Security Testing and Monitoring*. 15–21. doi:10.1145/2898375.2898381

[20] Martin Matousek, Radoslav Holubek, and Vladimir Chrenko. 2021. Applying Threat Analysis and Risk Assessment According to ISO/SAE 21434 in the Automotive Industry. In *2021 IEEE 44th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 1117–1122. doi:10.23919/MIPRO52101.2021.9597014

[21] Jacopo Mauro, Michael Nieke, Christoph Seidl, and Ingrid Chieh Yu. 2016. Context Aware Reconfiguration in Software Product Lines. In *Proceedings of the 10th International Workshop on Variability Modelling of Software-Intensive Systems* (Salvador, Brazil) *(VaMoS '16)*. Association for Computing Machinery, New York, NY, USA, 41–48. doi:10.1145/2866614.2866620

[22] Richard May, Christian Biermann, Andy Kenner, Jacob Krüger, and Thomas Leich. 2024. A Product-Line-Engineering Framework for Secure Enterprise-Resource-Planning Systems. *Procedia Computer Science* 239 (2024), 1619–1626. doi:10.1016/j.procs.2024.06.338 CENTERIS – International Conference on ENTERprise Information Systems / ProjMAN - International Conference on Project MANagement / HCist - International Conference on Health and Social Care Information Systems and Technologies 2023.

[23] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. Feature traceability in feature models and configurations. In *Mastering SoftwareVariabilitywithFeatureIDE*. Springer International Publishing, 73–80. doi:10.1007/978-3-319-61443-4_7

[24] Rodrigo André Ferreira Moreira, Wesley K. G. Assunção, Jabier Martinez, and Eduardo Figueiredo. 2022. Open-source software product line extraction processes: the ArgoUML-SPL and Phaser cases. *Empir. Softw. Eng.* 27, 4 (2022), 85. doi:10.1007/s10664-021-10104-3

[25] Julia Mucha, Andreas Kaufmann, and Dirk Riehle. 2024. A systematic literature review of pre-requirements specification traceability. *Requirements Engineering* 29 (2024), 119–141. doi:10.1007/s00766-023-00412-z

[26] National Institute of Standards and Technology. 2011. *Managing Information Security Risk: Organization, Mission, and Information System View.* Special Publication 800-39. NIST. https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-39.pdf

[27] Sven Peldszus, Daniel Strüber, and Jan Jürjens. 2018. Model-Based Security Analysis of Feature-Oriented Software Product Lines. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE)*. xx–xx. doi:10.1145/3278122.3278126

[28] M. Rosenmüller, N. Siegmund, S. Apel, et al. 2011. Flexible feature binding in software product lines. *Automated Software Engineering* 18, 2 (2011), 163–197. doi:10.1007/s10515-011-0080-5

[29] Jocelyn Simmonds, Daniel Perovich, María Cecilia Bastarrica, and Luis Silvestre. 2015. A megamodel for Software Process Line modeling and evolution. In *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015*. 406–415. doi:10.1109/MODELS.2015.7338272

[30] George Spanoudakis and Andrea Zisman. 2004. Software traceability: A roadmap. *Handbook of Software Engineering and Knowledge Engineering* 3 (2004), 395–428. doi:10.1142/9789812775245_0014

[31] Sagar Sunkle, Deepali Kholkar, and Vinay Kulkarni. 2015. Toward Better Mapping between Regulations and Operations of Enterprises Using Vocabularies and Semantic Similarity. *Complex Systems Informatics and Modeling Quarterly* (12 2015). doi:10.7250/csimq.2015-5.04

[32] Faiz Ul Muram, Barbara Gallina, and Samina Kanwal. 2019. A Tool-Supported Model-Based Method for Facilitating the EN50129-Compliant Safety Approval Process. In *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification*, Simon Collart-Dutilleul, Thierry Lecomte, and Alexander Romanovsky (Eds.). Springer International Publishing, Cham, 125–141.

A Research Methods

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009