Feature-based software architecture analysis to identify safety and security interactions

Priyadarshini*†, Simon Greiner*, Maike Massierer*, Oum-El-Kheir Aktouf†
*Robert Bosch GmbH, Cross Domain Computing Solutions, Abstatt, Germany
Email:{priyadarshini.priyadarshini,simon.greiner,maike.massierer}@de.bosch.com
†Univ. Grenoble Alpes, Grenoble INP, LCIS, Valence, France
Email: {priyadarshini.-,oum-el-kheir.aktouf}@lcis.grenoble-inp.fr

***Abstract*—In the automotive domain, feature-based software architecture is a widely used software design method to produce cost efficient and reusable software. With increasing complexity of automotive systems and the shift towards automated driving, safety and security measures become even more crucial for these systems. However, safety and security functionalities can undermine each other if they interact in unintended ways.**

We propose the novel method *FIISS* for automatic identification of interactions between safety and security features in UML models. We evaluate our implementation of the method by applying it to a real-world component for autonomous driving. We show that the method is able to identify unintended interactions while providing only few false positive findings. Thus, we see that our method can be applied to real-world UML system designs without modifying the underlying models and without applying specialized UML profiles.

***Index Terms*—model-based software architecture analysis, feature interaction, feature-based software engineering, functional safety feature, cybersecurity feature, unified modeling language (UML), autonomous vehicles**

I. Introduction

Due to the shift towards assisted and autonomous driving, the automotive industry faces the enormous challenge of ensuring safety and security of autonomous vehicles. Automotive standards such as ISO 26262 [1] for functional safety, ISO/PAS 21448 [2] for safety of the intended functionality, and ISO/SAE 21434 [3] for cybersecurity specify precise engineering requirements for electrical and electronic systems in road vehicles. This results in a growing number of safety and security requirements from original equipment manufacturers, and thus more safety and security features are being integrated into automotive systems.

An automotive security feature is a hardware or software measure that protects an automotive system against attacks from malicious actors, whereas an automotive safety feature aims to prevent safety hazards due to random errors, system failures or functional insufficiencies of the intended functionality. Both disciplines are highly specialized and require well-trained engineers to perform safety and security engineering according to their respective standards, methods, and development processes. However, communication and exchange between these disciplines is often limited due to insufficient awareness about safety and security interactions and the fact that safety processes are often more mature than the more recently introduced security processes.

In feature-based system development, features developed independently of each other can lead to unintended feature interactions when integrated into the system [4], [5]. In a safety-critical autonomous system, such interactions can make the system unsafe for the user. If, for example, a firewall detects unauthorized messages sent to the vehicle, it may discard these messages. However, the loss of a message may be safety-critical, e.g. for the braking functionality. A safety monitor could be in place to ensure that every received message is processed without exception. In this case, the functionality of the firewall is rendered useless. Notice that feature interactions may also exhibit synergies for co-design and co-development and improve the system performance.

In our experience, unintended interactions between safety and security features are identified by manual analysis of safety and security concepts and regular discussions between system, safety, and security architects. However, these are time-consuming activities, and experts trained in both safety and security engineering able of identifying safety and security interactions are rare. An exploratory survey [6] observes that the majority of automotive organizations do not consider interdependencies between safety and security requirements.

In this paper, we propose the novel method *FIISS – Feature Interaction Identification between Safety and Security* for the detection of interactions between safety and security features, where these features are modeled using the Unified Modeling Language (UML) [7]. FIISS analyzes the architectural and behavioral specification of a system represented in UML. It does not require the use of a dedicated UML profile, so that it can be practically applied to a large range of projects, including real-world systems. Hence our method can be used early during development, thus reducing costs compared to detecting problems late in a project.

FIISS is the first method to identify intended and unintended interactions between software safety and security features by analyzing the UML based software architectural model of a system. Our second contribution is the FIISS tool, which automatically extracts interactions between safety and security relevant software components, thereby significantly reducing the manual analysis required to identify these interactions.

The remainder of this paper is organized as follows. In Section II, we introduce core terminology on safety, security, features, and their interactions. Section III introduces FIISS

979-8-3503-9749-9/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSA56044.2023.00010

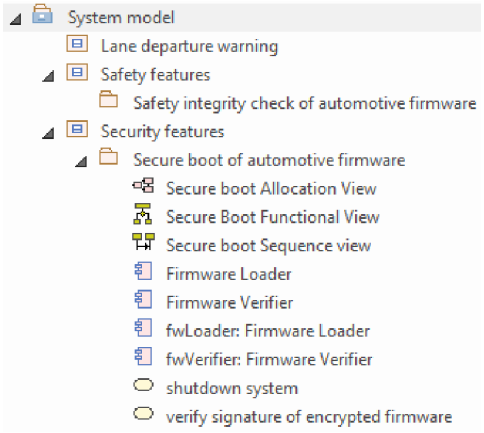

Fig. 1. Sample features in the software architecture model of a system

and describes the steps of the method. In Section IV, we present the most relevant design decisions concerning our implementation of FIISS as a tool. In Section V, we evaluate our method by applying it to a real-world driver assistance system and discuss the outcome of this evaluation. We compare our method to related work in Section VI. Finally, we conclude and discuss future work in Section VII.

II. Preliminaries

This section describes the terminology and underlying concepts used in this paper. For better readability, we format `UML elements this way` and SEMANTIC ELEMENTS extracted from the UML model LIKE THIS. We refrain from the highlighting when the readability of the text suffers.

A. Feature

We refer to a FEATURE as a distinct function that satisfies the specification of an automotive system. The function itself (such as braking or redundant communication) may or may not be accessed directly by the system user, i.e. the driver. In the UML model of the software architecture, `packages` represent features and contain the structural and behavioral specification of the feature.

A `package` contains a group of related UML elements such as diagrams, `components` or other `packages`, as shown in Fig. 1. Alternatively, a feature can be represented as a `component` with encapsulated behavior, such as an `operation`. In this work, we consider it a `package`, since this is a common modeling style in automotive projects.

Within a `package`, the functions of the corresponding feature are specified using `activities`, which can be decomposed into sub-functions. A sub-function can be represented as an `action` or an `activity`. The latter can then be decomposed further. Each `action` and `activity` is realized by one or more `components`; defining reusable components is a common practice in the automotive industry. The realization of `actions` and `activities` is modeled in UML using the `realization` relationship.

B. Security feature

Hamsini and Kathiresh [8] explain three layers of security for an automotive system, i.e. hardware security, software security, and network security features. We adopt their description of software security features from this categorization and use the term SECURITY FEATURE to refer to a software function or set of software functions designed to protect an asset from cyberattacks, where the asset is typically a (part of a) component in an automotive system. Common security features for electronic control units (ECU) include secure boot and secure communication. Even though these features may not be accessed directly by a vehicle occupant, they are essential to protect the system against external threats.

C. Safety feature

We use the term SAFETY FEATURE to refer to a software function or set of software functions designed to prevent or to detect and mitigate failures of a safety-critical system. Failures could result from random or design errors, e.g. a software bug. Typical safety features for an ECU include software lockstep, memory partitioning, and timing monitoring. These features may not be directly accessible by a vehicle occupant but contribute to overall system safety.

D. Interactions between safety and security features

Safety and security features are usually developed independently of each other, and dependencies between them can easily be missed. Such dependencies can include output of a security feature as a direct or transitive input to a safety feature, or security functionality triggering a reaction that is not allowed from a safety standpoint, since the security functionality is not automotive safety integrity level (ASIL) rated. If such dependencies are left undetected or unaddressed, this can lead to unintended system behavior, loss of functions, performance issues, or failure to use synergies between features. In this paper, we refer to such dependencies between safety and security features as INTERACTIONS.

III. The FIISS method

The FIISS method identifies potential unintended interactions between specific safety and security features in the architectural design phase of a system. The interactions found by FIISS then have to be reviewed by safety and security domain experts to identify unintended specification of behavior.

Afterwards, problematic interactions such as inconsistent system reactions to safety and security events, runtime performance issues, etc. should be resolved. Other interactions, such as synergies, can be useful to reduce system complexity or increase the efficiency of system components. During system architecture validation, the output of FIISS can serve as an input for system test engineers.

FIISS consists of three steps as shown in Fig. 2. First, we identify architectural elements, e.g. activities and components, related to safety and security features. In the second step, we analyze the communication between safety and security

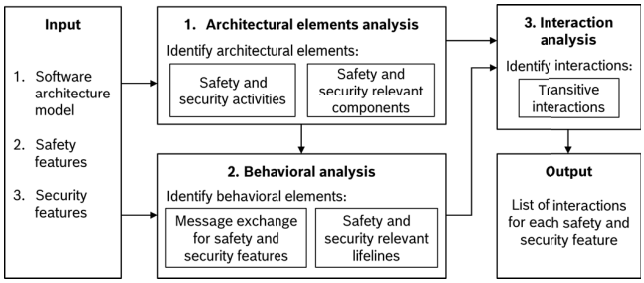

Fig. 2. Overview of the FIISS method

relevant components from the first step according to the behavioral specification in the UML sequence diagrams. Finally, we combine the results of the architectural and behavioral analysis to identify interactions between safety and security relevant components due to safety and security features.

A. Running example

As a running example, we use the software architecture model of a driver assistance ECU performing lane departure warning as shown in Fig. 1. Several software safety and security features are integrated in this ECU, e.g. a safety error handler and secure flashing of firmware.

To explain each step of FIISS in detail, we present the UML diagrams of the security feature *secure boot of automotive firmware* [9], [10], which protects the ECU against loading malicious firmware. To ensure integrity, authenticity, and confidentiality, the firmware is encrypted and signed with a digital signature. When the ECU loads the firmware, it must therefore first check the digital signature, and if this is successful, it can decrypt and load the firmware.

Similarly, an example of a safety feature in this ECU is *safety integrity check of automotive firmware* [11]. It verifies the integrity of firmware, e.g. by cyclic redundancy checks.

B. Input to FIISS

The first input to FIISS is a UML software architectural model of the system to be analyzed. The second and third inputs are a list of security and safety features respectively. These lists are obtained by identifying these features in the software architectural model. We assume that the architectural framework used for modeling the system facilitates to distinguish models of safety and security features from models of other features. Aligning the software architecture and the corresponding software design of the system being analyzed is not in the scope of this work.

Alternatively, a feature could be marked as safety or security feature using a dedicated UML profile, e.g. by applying a specific stereotype. We deliberately avoid the use of specific UML profiles, since in our experience, highly specialized profiles are often not used consistently or correctly in real-world projects, when modeling the functionality of the system is the main goal.

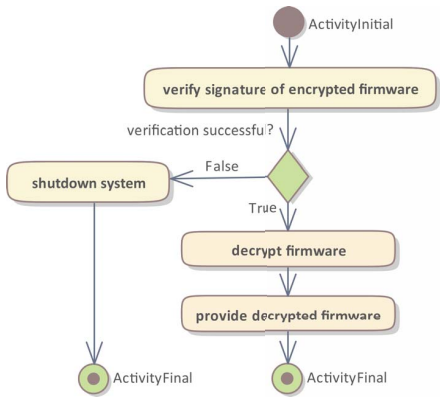

Fig. 3. Activity diag. of security feature *secure boot of automotive firmware*

C. Architectural elements analysis

In the first step, FIISS identifies architectural UML elements which are related to security and safety features of the system under consideration.

1) Safety and security activities: FIISS identifies activities which represent functions or sub-functions of safety and security features.

A SAFETY ACTIVITY of a SAFETY FEATURE is an `activity` or `action` inside the `package` representing the SAFETY FEATURE. Analogously, a SECURITY ACTIVITY of a SECURITY FEATURE is an `activity` or `action` inside the `package` representing the SECURITY FEATURE.

For example, in Fig. 1, the activities *shutdown system* and *verify signature of encrypted firmware* contained in the package *secure boot of automotive firmware* are security activities. These activities correspond to the activity diagram *Secure boot Functional View* as shown in Fig. 3.

2) Safety and security relevant components: FIISS identifies components that realize the safety and security activities found in the previous step.

A `component` is RELEVANT for a SAFETY FEATURE sf if it realizes a SAFETY ACTIVITY. We say a `component` c is SAFETY RELEVANT for SAFETY FEATURE sf if there exists a SAFETY ACTIVITY safact of sf and a `realization` relationship r in the model such that c is the source element of r and safact is the target element of r. Further, we say a component c is SAFETY RELEVANT if there exists a SAFETY FEATURE sf such that c is SAFETY RELEVANT for sf. Similarly, a component is SECURITY RELEVANT for a SECURITY FEATURE if it realizes a SECURITY ACTIVITY of this feature.

We illustrate this with an example as shown in Fig. 4. The component *Firmware Verifier* is security relevant for the feature *secure boot of automotive firmware* because it realizes the security activities *decrypt firmware* and *verify signature of encrypted firmware* of this security feature.

A component can be safety or security relevant for more than one feature. This is the case when a component realizes

several security or safety activities. Further, a component can be safety and security relevant at the same time.

D. Behavioral analysis

During behavioral analysis, FIISS identifies behavioral specifications in the form of message exchanges that are specified by a safety or a security feature. Further, FIISS identifies safety and security relevance of lifelines represented in sequence diagrams in relation to the architecture of the system.

1) Message exchange for safety and security features: To analyze the interactions between safety and security features, FIISS requires the specification of the behavior of the features as represented in the model. We concentrate here on the behavioral specification in the form of message exchange in sequence diagrams.

In this step, FIISS identifies a set of message exchanges for each safety and security feature. We use the term MESSAGE EXCHANGE for a triple consisting of a source lifeline, a sequence or message, and a destination lifeline. We say a MESSAGE EXCHANGE (l_1, m, l_2) is a MESSAGE EXCHANGE for a SAFETY or SECURITY FEATURE sf if there exists a `sequence` or `message` m in the `package` of sf such that l_1 is the `source lifeline` of m and l_2 is the `destination lifeline` of m.

An example is shown in Fig. 5. It shows a message exchange for the *secure boot* security feature, where the source lifeline is *fwVerifier*, the message is *provideDecryptedFirmware()*, and the destination lifeline is *fwLoader*.

2) Safety and security relevant lifelines: In this step, FIISS examines safety and security relevance of lifelines, i.e. whether or not the sending or receiving of a message can influence the behavior defined by a safety or security feature.

A lifeline is safety relevant for a feature if it is used in a sequence diagram of the feature (i.e. the lifeline is involved in the interaction due to the respective feature) and the lifeline represents an instance of a safety relevant component. More formally, we say a `lifeline` l is SAFETY RELEVANT for a SAFETY or SECURITY FEATURE sf if there exists a MESSAGE EXCHANGE (l_1, m, l_2) for sf with $l_1 = l$ or $l_2 = l$; and there exists a `component` c which is SAFETY RELEVANT and the `classifier` of l is c and c is not SECURITY RELEVANT.

Similarly, we define SECURITY RELEVANT LIFELINES. We say a `lifeline` l is SECURITY RELEVANT for a SAFETY or SECURITY FEATURE sf if there exists a MESSAGE EXCHANGE (l_1, m, l_2) for sf with $l_1 = l$ or $l_2 = l$; and there exists a `component` c which is SECURITY RELEVANT and the `classifier` of l is c and c is not SAFETY RELEVANT.

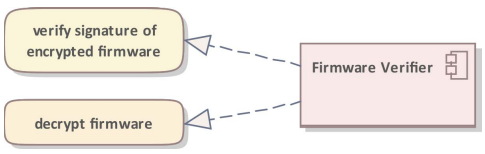

Fig. 4. Realization of security activities by relevant components

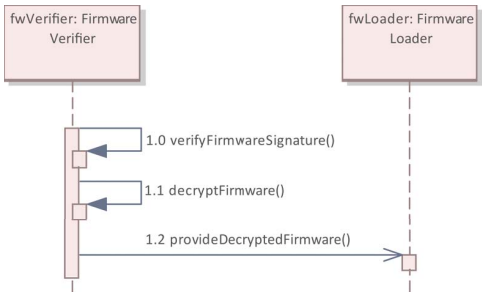

Fig. 5. Sequence diag. of security feature *secure boot of automotive firmware*

Finally, we define SAFETY-SECURITY RELEVANT LIFELINES. A `lifeline` l is SAFETY-SECURITY RELEVANT for a SAFETY or SECURITY FEATURE sf if there exists a MESSAGE EXCHANGE (l_1, m, l_2) for sf with $l_1 = l$ or $l_2 = l$; and there exists a `component` c which is SAFETY RELEVANT and the `classifier` of l is c; and it exists a `component` c with c being SECURITY RELEVANT and the `classifier` of l is c.

For example, in Fig. 5, *fwVerifier* is a security relevant lifeline since it is an instance of *Firmware Verifier*, which is a security relevant component.

Note that whether a lifeline is safety or security relevant for a specific feature depends on whether the lifeline is modeled in a sequence diagram of that feature. Whether the lifeline is safety or security relevant can be due to the security or safety relevance of the associated component for any safety or security feature. By defining safety and security relevance of lifelines this way, FIISS can identify interactions between safety and security components, even if not all required information about the relevance of components is modeled within the package that models the feature. Alternatively, the relevant lifelines can be identified through use of UML extensions that explicitly specify non-functional properties such as reliability and performance [12].

E. Interaction analysis

FIISS can now identify interactions in the form of message exchanges between safety and security features. If a message exchange involves a safety relevant component and a security relevant component, this exchange can potentially be problematic, since a safety functionality might undermine the correct functionality of a security feature or vice versa.

1) Transitive interaction identification: Direct interactions between components can be easily identified by analyzing message exchanges shown in sequence diagrams, or by just iterating through the message exchange set defined above. However, an interaction could be more subtle by involving other components in between the two relevant components which may be neither safety nor security relevant or both safety and security relevant. Therefore, FIISS also identifies transitive message exchanges defined by a feature.

For the safety or security feature being analyzed, FIISS can now identify interactions from its safety relevant lifelines to its security relevant lifelines and vice versa. When finding

Algorithm 1 Extract interactions between lifeline sets

```
for all safety or security relevant features sf do
    GET_INTER(sf, saf lls, sec lls)
    GET_INTER(sf, sec lls, saf lls)
    GET_INTER(sf, saf lls, saf-sec lls)
    GET_INTER(sf, sec lls, saf-sec lls)
    GET_INTER(sf, saf-sec lls, saf lls)
    GET_INTER(sf, saf-sec lls, sec lls)
```

Algorithm 2 Find paths in the interaction graph of a feature

```
function GET_INTER(feature sf, lifelines_1, lifelines_2)
    interactions ← emptyset
    for all l_1 in lifelines_1 do
        for all l_2 in lifelines_2 do
            if exists path in G_sf from l_1 to l_2 then
                p ← identified path
                interactions ← interactions + (l_1, l_2, p)
    return interactions
```

interactions that involve safety-security relevant lifelines, it is typically unclear whether the interaction modeled by a lifeline is due to the realization of a safety or a security activity. For example, it is possible that a message sent by a lifeline is the safety reaction (e.g. a monitor reaction) of the component caused by an input which is related to a security functionality (e.g. intrusion detection). We still consider safety-security relevant lifelines in our analysis in order to investigate whether they lead to the identification of interactions which would otherwise be missed.

To obtain all interactions of safety and security relevant components, FIISS now extracts interactions for each safety and security relevant feature with different combinations of sets of lifelines, as depicted in Algorithm 1. For a more compact representation, we abbreviate safety relevant lifelines (*saf lls*), security relevant lifelines (*sec lls*), and safety-security relevant lifelines (*saf-sec lls*). For example, for the feature sf = *secure boot of automotive firmware*, we want to find interactions from its *sec lls*, i.e. [fwVerifier] to its *saf-sec lls*, i.e. [fwLoader].

To identify interactions specified by a feature from elements in one set of lifelines to elements in another set of lifelines, FIISS finds paths in the interaction graph G_{sf} of a feature, as shown in Algorithm 2. The graph G_{sf} is defined by the set of all MESSAGE EXCHANGES of the SAFETY or SECURITY FEATURE sf identified in Section III-D1. FIISS interprets message exchanges specified by a feature as edges in this graph, where each lifeline is represented by a node and the message is represented by a directed edge. We say the SAFETY or SECURITY FEATURE sf specifies an INTERACTION from a `lifeline` l_1 to l_2 if there exists a path from l_1 to l_2 in G_{sf}.

We say the path p as used in the definitions above is an INTERACTION PATH. If the direction of an interaction is not important, we also say a safety or security feature sf specifies an interaction between two lifelines l_1 and l_2, if sf specifies an interaction from l_1 to l_2 or from l_2 to l_1. For example, FIISS identifies the interaction path p = [(*fwVerifier, decryptFirmware(), fwVerifier)*, (*fwVerifier, provideDecryptedFirmware(), fwLoader*)] between the source lifeline l_1 = *fwVerifier* and the destination lifeline l_2 = *fwLoader*. The interaction path p shows that *secure boot of automotive firmware* specifies an interaction from the security relevant component *Firmware Verifier* to the safety relevant component *Firmware Loader*. As FIISS also finds transitive interactions, the interaction path p can contain interactions modeled not only in one but multiple sequence diagrams.

2) Output preparation: The output of FIISS is a list of interaction paths for each safety and security feature. For every lifeline in the listed interaction paths, the output provides information on the safety and security relevance and corresponding safety and security activities with related features.

IV. IMPLEMENTATION

In this section, we describe the FIISS tool, an implementation of FIISS. We first describe the input format of the model and then the algorithms to extract architectural and behavioral elements of interest from the model. Then we present the steps of identifying interaction paths of safety and security relevant components in the behavioral specification of each safety and security feature.

A. Input

The input to the FIISS tool is a UML model in XML Metadata Interchange (XMI) [13] format, i.e. an XML file format commonly used to store model design information. XMI identifiers are unique identifiers that identify XML elements contained in the XMI file. Our second and third inputs are two lists of XMI identifiers that identify packages representing the security and safety features, respectively.

B. Extracting static architectural elements

The first step is to parse the input XMI file using the *lxml* Python library, which is commonly used to process XML files. Algorithm 3 describes the extraction of safety and security relevant components for each safety and security feature based on the explanation in Section III-C. First the function GET_ACTIVITES extracts XML elements of type `activity` and `action` by iterating through the package of the feature. Then for each extracted activity and action, the function GET_COMPONENTS iterates through the entire parsed model to find XML elements of type `realization` having an attribute *supplier* with XMI ID of the activity or action. From the identified XML elements, the algorithm extracts the attribute *client*, i.e. the XMI ID of the component.

For example, for the feature sf = *secure boot of automotive firmware*, the ComponentList obtained is [Firmware Verifier, Firmware Loader]. Fig. 4 shows the realization dependencies only for *Firmware Verifier* but not for other components due to space restrictions.

In this way, we retrieve a list of components for each safety and security feature. Once the FIISS tool extracts the

Algorithm 3 Get security/safety relevant comps per feature

```
1: function GET_RELEVANT_COMPONENTS(feature sf)
2:     ComponentList ← emptylist
3:     ActivityList ← GET_ACTIVITES(sf)
4:     for all activity in ActivityList do
5:         comp ← GET_COMPONENTS(activity)
6:         ComponentList ← ComponentList + comp
7:     return ComponentList
```

Algorithm 4 Get behavioral specification for each feature

```
1: function GET_BEHAVIORAL_SPEC(feature sf,
       SafetyCompList, SecurityCompList)
2:     MsgTripleList ← GET_MESSAGES(feature sf)
3:     LifelineList ← GET_LIFELINES(feature sf)
4:     saf_lifelines, sec_lifelines, saf-sec_lifelines ←
           GET_RELEVANT_LIFELINES(LifelineList,
           SafetyCompList, SecurityCompList)
5:     return saf_lifelines, sec_lifelines, saf-sec_lifelines
```

components for all safety features, it stores all safety relevant components in the list *SafetyCompList*. Similarly it stores all the security relevant components in the list *SecurityCompList*.

C. Extracting behavioral elements

Algorithm 4 extracts behavioral elements of safety and security features from the parsed model as explained in Section III-D. First the function GET_MESSAGES stores the message exchanges in the *MsgTripleList* by extracting the XML elements of type `message` from the package of the feature. Then the function GET_LIFELINES stores the lifelines of each feature in the *LifelineList* by extracting XML elements which represent lifelines in sequence diagrams, such as lifeline, component or actor. Next the function GET_RELEVANT_LIFELINES checks if the elements of the *LifelineList* are present in *SafetyCompList* or *SecurityCompList* or both. Based on this check, the algorithm returns safety, security and safety-security relevant lifelines for the given feature. For example, Algorithm 4 returns *saf_lifelines* = [], *sec_lifelines* = [fwVerifier] and *saf-sec_lifelines* = [fwLoader] for the feature sf = *secure boot of automotive firmware*.

D. Finding interaction paths

To find interaction paths between safety and security relevant lifelines, the FIISS tool creates a directed graph using the *MultiDiGraph* class of the *networkx* [14] Python library. We choose *MultiDiGraph* to create directed graphs because this allows multiple edges between nodes as well as self loops, i.e. an edge of a node going to itself, which is helpful to represent different elements modeled in a sequence diagram. For each feature, the tool creates a graph G_{sf} in which a node represents a lifeline in the *LifelineList* and an edge represents a message in the *MsgTripleList*. With *saf_lifelines*, *sec_lifelines* and *saf-sec_lifelines* as inputs, the FIISS tool calculates interactions between them based on Algorithm 1 in Section III-E.

E. Output format

The FIISS tool presents the output, i.e. interaction paths for each safety and security feature, in tabular form. Each table has 6 columns and describes an interaction path from a source to a destination lifeline as shown in Table I. The first column *Lifeline: lifelineType* contains the name and type of each lifeline in the identified path, the first being the source and the last being the destination lifeline. The second column *Message* contains the message that the corresponding lifeline sends. The third column indicates the *security relevance* of the lifeline, and the fourth column the *safety relevance*.

If the lifeline is security relevant, the fifth column shows the security activities this lifeline performs in square brackets and the security feature associated with these activities before the colon. Similarly, if the lifeline is safety relevant, the sixth column shows the safety activities and the safety feature associated with these activities.

V. Evaluation of the FIISS method

We evaluate FIISS by applying it to the model of a real-world automotive product. We first present some information about the product. Then we describe how FIISS is applied to this product and present the resulting output. We evaluate the output by reviewing the identified interactions with domain experts to assess whether they are actual and correct interactions, i.e. true positives in the model. Finally, we discuss the results of the evaluation.

A. System for evaluation

We apply FIISS to the UML software architectural model of an ECU which provides functionalities for autonomous and assisted driving. The ECU is a real-world product used in series vehicles and sold as a product. The model was created during product development using Enterprise Architect, a common system and software modeling tool in the automotive industry. The software architecture model contains 3209 classes, 567 components, and 8769 diagrams. The UML software architectural model of the ECU contains 22 security features and 48 safety features. We excluded 2 security features and 4 safety features from the evaluation because the model does not contain sequence diagrams for these features. We applied FIISS to the model as provided by the product development team without modification.

B. Application of FIISS

There are three steps in the FIISS application process. First, we provide the model in XMI format to the FIISS tool. Second, we apply the FIISS method by running the tool. Finally, we obtain the output of FIISS for evaluation.

1) Prepare input: In the first step, the input is prepared by exporting the UML architecture of the system being analyzed in XMI format. In the XMI file, the XMI IDs of the packages representing safety and security features are identified to create two lists of XMI IDs, one for safety and another for security features, respectively. This step is performed with the support of responsible safety and security engineers for the ECU.

TABLE I
IDENTIFIED INTERACTION PATH FOR THE FEATURE *secure boot of automotive firmware*

Lifeline: lifeline-Type	**Message**	**Security relevance**	**Safety relevance**	**Security relevant lifeline's activities and their associated features**	**Safety relevant lifeline's activities and their associated features**
fwVerifier: Firmware Verifier	verify-Firmware-Signature()	yes	no	'Secure Boot of automotive firmware': ['verify signature of encrypted firmware', 'shutdown system', 'decrypt firmware', 'provideDecryptedFirmware', 'configure security parameters']	-
fwVerifier: Firmware Verifier	decrypt-Firmware()	yes	no	'Secure Boot of automotive firmware': ['verify signature of encrypted firmware', 'shutdown system', 'decrypt firmware', 'provideDecryptedFirmware', 'configure security parameters']	-
fwVerifier: Firmware Verifier	provide-Decrypted-Firmware()	yes	no	'Secure Boot of automotive firmware': ['verify signature of encrypted firmware', 'shutdown system', 'decrypt firmware', 'provideDecryptedFirmware', 'configure security parameters']	-
fwLoader: Firmware Loader	verify-Firmware-Integrity()	yes	yes	'Secure Boot of automotive firmware': ['verify signature of encrypted firmware', 'shutdown system', 'decrypt firmware', 'provideDecryptedFirmware', 'configure security parameters']	'Safety integrity check of automotive firmware': ['verify integrity of firmware', 'initiate safe state', 'start boot']
fwLoader: Firmware Loader	-	yes	yes	'Secure Boot of automotive firmware': ['verify signature of encrypted firmware', 'shutdown system', 'decrypt firmware', 'provideDecryptedFirmware', 'configure security parameters']	'Safety integrity check of automotive firmware': ['verify integrity of firmware', 'initiate safe state', 'start boot']

Note: This interaction path is adapted slightly to protect the intellectual property contained in the ECU.

2) Apply FIISS: The XMI file and two lists of XMI IDs are fed as inputs to the FIISS tool, which applies FIISS automatically. We run the FIISS tool to obtain the output, i.e. interactions between relevant lifelines for each safety and security feature.

3) Obtain FIISS output: The output of the FIISS tool is a file containing interaction paths in tabular form as depicted in Table I and described in Section IV-E. A summary of the number of security and safety features for which the FIISS tool found interaction paths is shown in Table II.

The notation saf $\leftrightarrow$ sec in the second column refers to interaction paths from security relevant to safety relevant lifelines and vice versa. The notation X $\leftrightarrow$ saf-sec refers to interaction paths from either security or safety relevant lifelines represented as X to safety-security relevant lifelines and vice versa. X $\leftrightarrow$ X includes interaction paths from both cases, i.e. saf $\leftrightarrow$ sec and X $\leftrightarrow$ saf-sec. These notations will be used in the remainder of this paper to refer the different kinds of interaction paths. In total, the FIISS tool identified interaction paths for 8 out of 20 security features and 12 out of 44 safety features.

Since we apply our tool to a real-world product, we cannot name the safety and security features in this paper, and even less the interactions between them, as this would reveal far-reaching details about design decisions and properties of the product. In order to protect this intellectual property, we anonymize the features, except for one example for security and safety each, i.e. *secure boot of automotive firmware* and *safety integrity check of automotive firmware*.

TABLE II
SAFETY AND SECURITY FEATURES WITH IDENTIFIED INTERACTIONS

	saf $\leftrightarrow$ sec	X $\leftrightarrow$ saf-sec	X $\leftrightarrow$ X
Safety and security features	**Features with interactions**	**Features with interactions**	**Features with interactions**
Security features (20)	6	8	8 out of 20
Safety features (44)	5	8	12 out of 44

C. Evaluation of FIISS output

We evaluate FIISS by manually analyzing each feature for which an interaction path was identified, for whether an interaction path represents a *true positive* or a *false positive* interaction. An identified interaction is said to be a *true positive* if it is a correctly identified problematic or synergetic interaction in the model. In contrast, an identified interaction is said to be a *false positive* if it is an incorrectly identified interaction, i.e. an interaction in which the functionalities of the interacting features do not exhibit any potential issues or synergies with respect to each other. We performed this manual analysis together with safety, security, and architecture experts for the ECU in order to ensure correct interpretation of our findings.

In the following, we present the manual analysis approach as an example with a real finding. We analyze one of the 42 identified interaction paths of the security feature *secure boot of automotive firmware*. Table I shows one interaction path that the FIISS tool found for this feature.

1) Identify interacting safety and security features: In an interaction path of a safety or security feature sf, we identify the corresponding interacting features of sf, i.e. security features in the column *Security relevant lifeline's activities and their associated features* if sf is a safety feature or safety features in the column *Safety relevant lifeline's activities and their associated features* if sf is a security feature.

TABLE III
INTERACTION SUMMARY FOR EACH SECURITY FEATURE

Security features	saf ↔ sec				X ↔ saf-sec			
	Interaction paths	**Interacting safety features**	**True positives**	**False positives**	**Interaction paths**	**Interacting safety features**	**True positives**	**False positives**
Security feature 1	90	3	3	0	38	3	3	0
Security feature 2	63	3	2	1	38	3	2	1
Security feature 3	108	2	2	0	261	2	2	0
Security feature 4	3	1	0	1	9	3	1*	2
Security feature 5	2	1	0	1	2	1	0	1
Security feature 6	4	1	0	1	12	3	1	2
Security feature 7	0	N/A	N/A	N/A	10	4	1 + 1*	2
Secure boot of automotive firmware	0	N/A	N/A	N/A	42	1	1	0

Note: (∗) marks interactions that could not be classified as true or false positives because the models were not sufficiently detailed.
Abbreviation: N/A = not applicable

TABLE IV
INTERACTION SUMMARY FOR EACH SAFETY FEATURE

Safety features	saf ↔ sec				X ↔ saf-sec			
	Interaction paths	**Interacting security features**	**True positives**	**False positives**	**Interaction paths**	**Interacting security features**	**True positives**	**False positives**
Safety feature 1	161	2	2*	0	0	N/A	N/A	N/A
Safety feature 2	224	1	1	0	0	N/A	N/A	N/A
Safety feature 3	3	1	1	0	0	N/A	N/A	N/A
Safety feature 4	132	1	1	0	0	N/A	N/A	N/A
Safety feature 5	12	1	1	0	10	2	2*	0
Safety feature 6	0	N/A	N/A	N/A	25	2	2*	0
Safety feature 7	0	N/A	N/A	N/A	4	5	1 + 4*	0
Safety feature 8	0	N/A	N/A	N/A	3	1	1*	0
Safety integrity check of automotive firmware	0	NA	NA	NA	12	1	1	0
Safety feature 10	0	N/A	N/A	N/A	34	2	1	1
Safety feature 11	0	N/A	N/A	N/A	4	1	1*	0
Safety feature 12	0	N/A	N/A	N/A	1	1	1*	0

In the interaction path of *Secure boot of automotive firmware* as shown in Table I, this security feature interacts with the safety feature *Safety integrity check of automotive firmware*. The interacting safety features for each security feature are presented in Table III, and the security features for each safety feature in Table IV. We omit features for which no interaction path was found due to space restrictions.

2) Identify true and false positive interactions: We interpret an interaction represented by an interaction path of a safety or a security feature sf by checking the message sequence in the column *Message* with respect to the activities performed by the safety and security relevant lifelines in the interaction path. Further, to understand the activities performed by the relevant lifelines, we refer to the activity diagrams of the feature sf and the interacting features of sf.

We analyzed the activity diagrams of these interacting features. The lifeline *fwVerifier* shuts down the system if the condition *verification successful?* is false, as shown in Fig. 3. The lifeline *fwLoader* initiates a safe state of the system if the condition *integrity verification successful?* is false.

Our key observations from this analysis are the following. First, the integrity verification by *fwLoader* cannot occur if the signature verification by *fwVerifier* is unsuccessful. In this case, we see that the functionality of the safety relevant component may not be carried out due to the function performed by the security relevant component. In addition, the system reactions triggered by both components are not aligned. Typically, system reactions such as a shutdown must be

performed both safely and securely in safety-critical systems. Based on these observations, we consider this interaction to be a *true positive* interaction.

A summary of the number of safety and security features with true positive interactions are shown in Table V. Among the identified *features with interactions*, we found that 7 out of 8 security features and 12 out of 12 safety features have *true positive* interactions with one or more of their interacting features. We found that in some special cases, the model of the feature being analyzed or its interacting features were underspecified, and thus we had insufficient information to classify the interaction as a true or a false positive. Such findings led to further discussions and investigations by our experts. We treat these findings of FIISS as *true positives*, since they are candidates for true positive interactions that must be further investigated. Moreover, they must be modeled in more detail to ease implementation for developers.

D. Discussion of FIISS evaluation

FIISS finds problematic and synergetic interactions between safety and security features with few false positives. FIISS was able to identify interaction paths even for features with missing activity diagrams and realization relationships. This is possible because we can still identify safety and security relevant lifelines in the sequence diagrams of these features if they use software components that were marked as safety or security relevant in the models of other features.

Using the interaction summaries in Tables III and IV, we compared the number of true positive interactions for security and safety features respectively in two categories, namely saf $\leftrightarrow$ sec, which does not consider safety-security relevant lifelines, and X $\leftrightarrow$ saf-sec, which considers safety-security relevant lifelines. The comparison shows that we would miss 4 out of 7 security features with true positive interactions and 7 out of 11 safety features with true positive interactions if safety-security relevant lifelines were not taken into account. From this finding, we deduce that it is important to consider interactions involving safety-security relevant lifelines. Approaches to associate activity diagrams with sequence diagrams, such as [15], can be used to determine the safety and security relevance of messages whose source or destination lifeline is a safety-security relevant lifeline.

In some cases sequence diagrams were under-specified. For example, if an operating system appears as a lifeline in the sequence diagrams of both safety and security features, we cannot determine whether the involved interaction is a true or a false positive because the model does not specify whether the library, the kernel, or some other element of the operating system is performing the specified activity.

E. Performance of the FIISS tool

The FIISS tool takes 10 minutes and 67 seconds to calculate the interaction paths for the UML model we used for evaluation. The graphs created for the safety and security features in the product contain up to 13 nodes and 165 edges, and due to this moderate size, it is fast to compute the interaction paths in these graphs. We thus conclude that even for a large real-world model as the one used here, the computations required to execute FIISS are sufficiently fast.

F. Completeness and correctness of FIISS

We have covered various corner cases, such as considering different lifeline classifiers. However, the FIISS method is not complete in the sense that it is not able to identify all modeled interactions between features. For example, if a sequence diagram contains a link to another sequence diagram, message exchanges contained in the other diagram are currently not analyzed. It is very likely that other corner cases are also not covered by our method.

FIISS cannot find interactions involving some safety and security features that are provided by third party suppliers. These features are not modeled in detail because their implementation details are unknown to the integrating party. Often, additional details about these features can be requested from the suppliers for the purpose of safety and security analysis. These details can then be used in the sequence diagrams of these features, enabling FIISS to identify feature interactions.

The most important property to assess the usefulness of FIISS is its correctness. Although the output of FIISS does contain false positive findings, the evaluation shows that FIISS is useful in practice for real-world systems, since it finds interactions with few false positives in a real-world model.

G. Threats to validity

An internal threat to our evaluation approach is the differentiation between true positive and false positive feature interactions. It is based on the judgement of experts and is hence subject to bias. To minimize this bias, we have performed the evaluation with the support of multiple safety and security experts. An external threat to FIISS is that a comprehensive usability study has not yet been performed.

VI. Related work

The research fields that focus on problems similar to ours are the emerging topic of safety and security co-engineering and the relatively established domain of feature interactions in software engineering. The *feature interaction problem* has been studied extensively in the telecommunication domain but is a relatively new topic in the automotive domain [16], [17]. The automotive industry heavily relies on semi-formal notations such as UML for software specification [18], [19]. This motivated us to analyze the UML models of features in the architectural design phase.

In the safety and security co-engineering domain, the work typically focuses on the requirements phase, the risk assessment phase, or the design phase. Some contributions cover the entire product lifecycle. Gu, Lu and Li [20] propose a goal-based approach to extract safety and security requirements and resolve the conflicts between them. This approach requires a manual analysis of requirements for a system typically available in textual form. Sun et al. [21] use a formal framework to analyze conflicts in the requirements phase which requires

TABLE V
SAFETY AND SECURITY FEATURES WITH TRUE POSITIVE INTERACTIONS

Features	saf ↔ sec		X ↔ saf-sec		X ↔ X	
	Features with interactions	**Features with true positive interactions**	**Features with interactions**	**Features with true positive interactions**	**Features with interactions**	**Features with true positive interactions**
Security features (20)	6	3	8	6	8 out of 20	7 out of 8
Safety features (44)	5	5	8	8	12 out of 44	12 out of 12

a formal model to be specifically created (if it does not exist for other reasons). In contrast to these approaches, FIISS can be applied to UML models that usually exist already.

The AVES framework [22] defines a matrix-based approach which allows a structured, manual analysis of conflicts and synergies between safety and security requirements and measures. By applying this approach throughout the lifecycle of a product, an overview of conflicts can be maintained. FIISS focuses on the design phase of product development and can be used to provide input to these matrices. Amorim et al. [23] propose a co-engineering approach, where the systematic application of safety and security architecture patterns aims to avoid the introduction of conflicts between safety and security even before they could occur. In contrast to this approach, FIISS aims to automatically identify interactions after they have been introduced during the design.

Feature interaction is a challenge in automotive software engineering, since an automotive system incorporates hundreds of user functions and thousands of atomic software functions [24], [18]. An empirical study [25] analyzing the functional architecture of an automotive software system found that 85% of the analyzed vehicle features depend on each other. While many feature interactions are known and intended, others are unknown and unintended [26]. Wagner et al. [27] aim to minimize CO_2 emission by optimizing feature interactions at run-time for automotive combustion engines.

Approaches centric to combined safety and security risk assessment such as FACT graph [28] and BDMP dynamic formalism [29] can potentially bring safety and security features that may interact closer to each other. Some works [27], [30] focus on detecting feature interactions due to the interplay of physical forces, such as simultaneous requests to apply brakes and throttle. While these approaches address the feature interactions visible at user level, we focus on identifying subtle feature interactions in electronic control units (ECU) which may or may not be visible at user level.

Other approaches to identify feature interactions are based on formal methods, i.e. model checking [31], [32] and analysis of linear temporal logic (LTL) properties [33], [34]. These approaches require a detailed formal model of the system. They can provide mathematically proven correctness and completeness guarantees. Defining the required models is typically very time consuming and must be done by highly trained experts. In contrast, we use UML sequence diagrams, which are commonly created during system and software design.

VII. CONCLUSION AND FUTURE WORK

This paper presents *FIISS*, a novel method to identify interactions between safety and security features during the architectural design phase of a system by analysis of the UML model. The method analyzes the architectural structure of a system in order to identify components related to safety and security features. By assessing the dynamic behavior of these components, FIISS can identify interactions between safety and security features. The method does not require the application of specialized UML profiles or modeling techniques, which are rarely used in the automotive industry.

We validate FIISS by applying an implementation of the method to a software architecture model of a real-world automotive ECU. Our evaluation results show that 95% of the identified interactions are true positives, while only 5% are false positives. This false positive rate is sufficiently low for the method to be useful in practice. When manually checking each detected interaction, the effort spent on checking false positives is reasonable. Our evaluation also shows FIISS' ability to detect subtle and transitive interactions, which can be hard to identify by hand. Our method can serve as an important tool for component and system test engineers, especially during early phases of the development.

Sometimes sequence diagrams contain links referencing other sequence diagrams. FIISS does not follow such links. If such a link points to a sequence diagram that is not contained in the package of the feature under consideration, FIISS misses resulting potential interactions. In future work, FIISS can be extended to follow the links discussed here. This way, the completeness of the output can be increased.

In addition, we could consider the temporal order of messages in sequence diagrams in order to optimize the number of interaction paths the method finds and to ease the manual analysis performed by domain experts.

A taxonomy for categories of interactions is presented in [35]. An automatic categorization of interactions within this taxonomy is an interesting research question. Another open line of research is to follow other combinations of UML specification elements in order to identify interactions more precisely and to further simplify the manual analysis required.

ACKNOWLEDGMENT

This work is supported by the French National Research Agency in the framework of the "Investissements d'avenir" program (ANR-15-IDEX-02).

REFERENCES

[1] International Organization for Standardization, "ISO 26262: Road vehicles – Functional Safety," 2011.

[2] ——, "ISO 21448: Road vehicles – Safety of the intended functionality," 2022.

[3] ——, "ISO/SAE 21434: Road vehicles – Cybersecurity engineering," 2021.

[4] S. Apel and C. Kästner, "An overview of feature-oriented software development." *J. Object Technol.*, vol. 8, no. 5, pp. 49–84, 2009.

[5] B. Glas, C. Gebauer, J. Hänger, A. Heyl, J. Klarmann, S. Kriso, P. Vembar, and P. Wörz, "Automotive safety and security integration challenges," *Automotive-Safety & Security 2014*, 2015.

[6] M. Huber, M. Brunner, C. Sauerwein, C. Carlan, and R. Breu, "Roadblocks on the Highway to Secure Cars: An Exploratory Survey on the Current Safety and Security Practice of the Automotive Industry," in *Computer Safety, Reliability, and Security*, B. Gallina, A. Skavhaug, and F. Bitsch, Eds. Cham: Springer International Publishing, 2018, pp. 157–171.

[7] O. M. G. (OMG), "Unified Modeling Language, Version 2.5.1," OMG Document Number formal/2017-12-06. [Online]. Available: https://www.omg.org/spec/UML/2.5.1

[8] S. Hamsini and M. Kathiresh, "Automotive safety systems," in *Automotive Embedded Systems*. Springer, 2021, pp. 1–18.

[9] R. S. Devi, B. V. Kumar, P. Sivakumar, A. N. Lakshmi, and R. Tripathy, "Bootloader Design for Advanced Driver Assistance System," in *Software Engineering for Automotive Systems*. CRC Press, 2022, pp. 31–44.

[10] M. Schrötter, "Understanding and Designing an Automotive-Like Secure Bootloader," *RARC 2020*, p. 105, 2020.

[11] Y. Marathe, K. Chitnis, and R. Garg, "Boot time optimization techniques for automotive rear view camera systems," in *2016 IEEE International Conference on Consumer Electronics (ICCE)*. IEEE, 2016, pp. 77–80.

[12] A. Krishna and A. Gregoriades, "Extending UML with non-functional requirements modelling," in *Information Systems Development*. Springer, 2011, pp. 357–372.

[13] O. M. G. (OMG), "XML Metadata Interchange XMI Specification, Version 2.5.1," OMG Document Number formal/2015-06-07. [Online]. Available: https://www.omg.org/spec/XMI/2.5.1

[14] L. A. N. Laboratory. (2022) Networkx – network analysis in python. [Online]. Available: https://networkx.org/

[15] S. E. Viswanathan and P. Samuel, "Automatic code generation using unified modeling language activity and sequence models," *Iet Software*, vol. 10, no. 6, pp. 164–172, 2016.

[16] T. Bowen, F. Dworack, C. Chow, N. Griffeth, G. Herman, and Y.-J. Lin, "The feature interaction problem in telecommunications systems," in *Seventh International Conference on Software Engineering for Telecommunication Switching Systems, 1989. SETSS 89.*, 1989, pp. 59–62.

[17] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec, "Feature interaction: a critical review and considered forecast," *Computer Networks*, vol. 41, no. 1, pp. 115–141, 2003.

[18] A. Pretschner, M. Broy, I. H. Kruger, and T. Stauner, "Software engineering for automotive systems: A roadmap," in *Future of Software Engineering (FOSE '07)*, 2007, pp. 55–71.

[19] G. Bahig and A. El-Kadi, "Formal Verification Framework for Automotive UML Designs," in *Proceedings of the 2nd Africa and Middle East Conference on Software Engineering*, ser. AMECSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 2127.

[20] T. Gu, M. Lu, and L. Li, "Extracting interdependent requirements and resolving conflicted requirements of safety and security for industrial control systems," in *2015 First International Conference on Reliability Systems Engineering (ICRSE)*. IEEE, 2015, pp. 1–8.

[21] M. Sun, S. Mohan, L. Sha, and C. Gunter, "Addressing safety and security contradictions in cyber-physical systems," in *Proceedings of the 1st Workshop on Future Directions in Cyber-Physical Systems Security (CPSSW09)*, 2009.

[22] G. Sabaliauskaite, L. S. Liew, and F. Zhou, "AVES – automated vehicle safety and security analysis framework," in *ACM Computer Science in Cars Symposium*, 2019, pp. 1–8.

[23] T. Amorim, H. Martin, Z. Ma, C. Schmittner, D. Schneider, G. Macher, B. Winkler, M. Krammer, and C. Kreiner, "Systematic pattern approach for safety and security co-engineering in the automotive domain," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2017, pp. 329–342.

[24] M. Broy, "Challenges in Automotive Software Engineering," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 3342.

[25] A. Vogelsang and S. Fuhrmann, "Why feature dependencies challenge the requirements engineering of automotive systems: An empirical study," in *2013 21st IEEE International Requirements Engineering Conference (RE)*, 2013, pp. 267–272.

[26] S. Apel, S. Kolesnikov, N. Siegmund, C. Kästner, and B. Garvin, "Exploring Feature Interactions in the Wild: The New Feature-Interaction Challenge," in *Proceedings of the 5th International Workshop on Feature-Oriented Software Development*, ser. FOSD '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 18.

[27] D. Wagner, H. Kaindl, S. Dominka, and M. Dübner, "Optimization of feature interactions for automotive combustion engines," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 2016, pp. 1401–1406.

[28] G. Sabaliauskaite and A. P. Mathur, "Aligning cyber-physical system safety and security," in *Complex Systems Design & Management Asia*. Springer, 2015, pp. 41–53.

[29] L. Pitre-Cambacds and M. Bouissou, "Modeling safety and security interdependencies with BDMP (Boolean logic Driven Markov Processes)," in *2010 IEEE International Conference on Systems, Man and Cybernetics*, 2010, pp. 2852–2861.

[30] Juarez Dominguez, Alma L., "Detection of Feature Interactions in Automotive Active Safety Features," Ph.D. dissertation, 2012. [Online]. Available: http://hdl.handle.net/10012/6701

[31] A. L. Juarez-Dominguez, N. A. Day, and J. J. Joyce, "Modelling Feature Interactions in the Automotive Domain," in *Proceedings of the 2008 International Workshop on Models in Software Engineering*, ser. MiSE '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 4550.

[32] A. P. Felty and K. S. Namjoshi, "Feature specification and automated conflict detection," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 12, no. 1, pp. 3–27, 2003.

[33] M. Calder and A. Miller, "Feature interaction detection by pairwise analysis of LTL properties – A case study," *Formal Methods in System Design*, vol. 28, no. 3, pp. 213–261, 2006.

[34] ——, "Using SPIN for feature interaction analysis – A case study," in *International SPIN workshop on Model Checking of Software*. Springer, 2001, pp. 143–162.

[35] M. Shehata, A. Eberlein, and A. O. Fapojuwo, "A taxonomy for identifying requirement interactions in software systems," *Computer Networks*, vol. 51, no. 2, pp. 398–425, 2007.