J Comput Virol Hack Tech manuscript No.
(will be inserted by the editor)

Verifying data secure flow in AUTOSAR models by static analysis

Received: date / Accepted: date

Abstract This paper presents an approach for enhancing the design phase of AUTOSAR models when security annotations are required. The approach is based on information flow analysis and abstract interpretation. The analysis evaluates the correctness of the model by assessing if the flow of data is secure with respect to causal data dependencies within the model. To find these dependencies an exhaustive search through the model would be required. Abstract interpretation is used as a trade-off between the precision and complexity of the analysis. The approach also provides annotated models without oversizing the set of annotations.

Keywords AUTOSAR · Security · Information flow · Static analysis

1 Introduction

Modern automotive electronics systems are real-time embedded systems running over networked Electronic Control Units (ECU) interconnected by wired networks such as the Controller Area Network (CAN) or Ethernet. Moreover, wireless connectivity is increasingly used for additional flexibility and bandwidth for features like key-less entry, diagnostics, and entertainment. This increased connectivity leads to an increasing number of potential cyber-security threats. Security in automotive systems is therefore becoming increasingly important and should be taken into account from the early stages of system design.

As part of recent extensions and developments, AUTOSAR [2], the reference standard for designing automotive systems, now offers a set of security-related services, which provides security functions such as encryption, integrity and authentication of messages exchanged over car networks. However, AUTOSAR does not provide any means to specify security requirements at the level of application components, but rather requires the application developers to directly use the standard security services. This is in contrast with the AUTOSAR approach for scheduling and communication. Application components are not allowed to use the basic software services for communication over network buses or the operating system services. Rather, high level specifications are provided and, based on them, code is automatically generated by tools to define the threads and invoke the appropriate network communication services. For this reason, the AUTOSAR component-based model has been extended by means of a set of *security annotations* that are statically assigned to components and links between components [13]. Security annotations allow us to specify trust levels on components and integrity and confidentiality requirements on communication links.

A proper security annotation depends on both the criticality of functionalities to be protected and the causal dependencies between data. Intuitively, if, for example, a safety-critical component requires the integrity of input data then we have to protect integrity along the whole path from data origin, e.g. sensors, to the component. Unfortunately, AUTOSAR only provides limited information about data causal dependencies. AUTOSAR provides information about ports each runnable (i.e. a functional unit running within a component) reads and writes. Such a coarse grain notion of data causal dependency may cause a redundant security annotation. Intuitively, this means that we may end up to protect certain data paths that do not actually influence a given critical functionality with a con-

sequent excessive utilisation of security-related services and consequent negative impact on performance and real-time constraints. This issue is particularly relevant in the automotive domain where software components run on resource-limited computer networks.

In this paper we propose a method to discover data dependencies with a finer level of granularity than AUTOSAR, by analysing the code of runnables. The analysis of data dependencies is based on abstract interpretation [16], a static analysis technique for the automatic extraction of information about the possible executions of programs.

A further contribution of the paper is the definition of the *Data secure flow* property of AUTOSAR models. *Data secure flow* verifies that, taking into consideration data causalities, the security annotations in the model are properly assigned. The data dependencies computed with our method can be exploited to strategically annotate an AUTOSAR model in such a way that the *Data secure flow* is verified, and the security services are efficiently used.

One of the advantages of our approach is that, being based on abstract interpretation, the analysis can be fully automated. Moreover, the analysis scales up, since runnables of AUTOSAR software components are analysed separately.

As a last contribution, the tool ADEPT (Autosar DEPendencies Tool) has been developed to support the analysis. The tool has been applied to a case study to show the application of the proposed method.

The paper is organised as follows. Section 2 reports on related work. Section 3 introduces the background on AUTOSAR models and information flow analysis. Section 4 provides a comprehensive description of the proposed approach. Section 5 describes the implementation of the method, and provides information on the developed tool. Section 6 shows the application of the approach to a case study.

2 Related work

This section reports on the research works about automotive security issues and information flow analysis.

2.1 Security

Recent research has shown that it is possible for external intruders to intentionally compromise the proper operation and functionality of modern automotive electronics systems. In [22], it has been demonstrated that if an adversary were able to communicate on one or more of a car internal network buses, then this capability could be sufficient to maliciously control critical components across the entire car.

The work [15] demonstrated that external attacks are indeed feasible and it also categorised external attack vectors as a function of the attacker ability to deliver malicious input via particular modes: indirect physical access, short-range wireless access, and long-range wireless access. Further remote attacks have been recently demonstrated in [37].

Security has been taken into account in the early phases of the development cycle of automotive electronics systems, both by enforcing software programming standards that prevent software defects that may enable cyber-attacks [15], as well as by implementing security mechanisms for secure communication [24,25], including software delivery, installation and flashing [1,35]. Factors like Required Resources and Required Know-How have been considered in the SAHARA (Security-Aware Hazard Analysis and Risk Assessment) method for defining threats criticality [27].

In [13,14] a set of modelling extensions has been defined to address AUTOSAR cyber-security requirements at design stage. Security requirements are modelled as stereotypes extending the AUTOSAR implementation provided by the IBM Rhapsody tool [18]. A similar approach has been used in SecureUML to model systems with role-based access control policies [26], and in umlsec to specify confidentiality properties of message communication [19]. Concepts and mechanisms that allow us to model confidentiality and authentication requirements at a higher abstraction level have been proposed in [32].

2.2 Data flow

Data flow in AUTOSAR models is analysed in the application configuration phase, where runnables must be grouped into tasks. Tasks are the unit of scheduling of the AUTOSAR operating system and they are executed in sequence. If a runnable reads data produced by another runnable, the first runnable cannot start until the second runnable finishes. The dependencies among runnables are computed by assuming that any communication implemented between two runnables represents a dependency [20].

The dependencies among runnables enable a correct parallel execution of runnables, so they must be respected even in the migration from single core to multi-core architectures [30], [20]. In [21] a tool for supporting parallel execution is developed. The tool executes data dependency analysis directly on AUTOSAR

models to detect critical dependencies. The static data dependency analysis approach is defined in [31].

All approaches above apply data-flow analysis to obtain the dependencies among runnables for identifying a proper execution order of runnables. The dependencies are computed by considering their accessed data. If any execution path of a runnable receives data on a port, the runnable depends on the runnable that sent such data.

In our work, the result of a data flow analysis, is the basis for checking the secure data flow in security annotated AUTOSAR models. For what concerns the data flow analysis, our approach differs from those mentioned above because we find dependencies at a finer granularity level: our iterative data flow analysis computes dependencies among data read from or written onto ports of the whole AUTOSAR model. Data secure flow property is computed by an algorithm that abstracts real values from data and considers only the security level of the data. The term "security level" is related to the security properties that data must satisfy. The higher the security level, the more properties have to be guaranteed. The abstract interpretation approach has been used for both the implementation of the data-flow analysis and the checking of secure flow property.

2.3 Secure flow

Data flow analysis is the basis for secure information flow in programs. The secure flow property in programs was first formulated in [11]. Successively, in [17], program certification was addressed, which statically checks secure information flow by inspecting the dependencies among variables in the program.

Works on static analysis techniques for information flow security in programs can be divided into type-based approaches and semantic-based approaches. In type-based approaches the security of a variables belongs to its type and secure information flow is checked by type systems [36,12]. An approach has been presented in [38] based on a continuation passing style translation of programs (continuations are used to handle implicit flows), while the work [9] handles secure information flow in object oriented languages. In semantic-based approaches, abstract interpretation is applied. For example, the work [29] presents a method based on denotational semantics, while the works [10] [8] are based on the operational semantics. In [23] an approach is presented based on axiomatic semantics, while the work [34] defines a method based on partial equivalence relations. The reader can refer to [33] for a survey.

The approach proposed in this paper relies on abstract interpretation of the operational semantics. The analysis is based on a transition system and thus has the advantage of being fully automatic. Our work differs from previous work because in AUTOSAR, we have both a set of component based modelling constructs, and a programming language used to describe the behaviour of runnable entities. Moreover, with respect to [10], data types and functions are included in the analysis.

3 Basic Concepts

This section provides an introduction to AUTOSAR and some basic knowledge about information flow analysis.

3.1 AUTOSAR

AUTOSAR is an open industry standard for automotive software architecture, founded in 2003 and developed by a partnership of automotive Original Equipment Manufacturers (OEMs), suppliers and tool vendors [2]. AUTOSAR provides both a standard language for the description of application components and their interfaces, and a methodology for the development process. A fundamental concept in AUTOSAR is the separation between application and infrastructure, see Figure 1. In particular, AUTOSAR defines a three-layered architecture consisting of:

- The Application layer
- The Runtime Environment (RTE) layer
- The Basic Software (BSW) layer

The Application Layer contains the Software Components (SWCs) developed for the automotive system functions by suppliers. The RTE layer is a middleware layer, automatically generated by tools and providing a communication abstraction for software components. Finally, the BSW layer provides basic services and basic software modules to software components. Within the BSW layer, AUTOSAR makes security mechanisms available to the developers in three different modules: a) the *Secure On-board Communication* (SecOC) module [7], which routes IPDUs (Interaction layer Protocol Data Units) with security requirements; b) the *Crypto Abstraction Library* (CAL) [5], which implements a library of cryptographic functions; and, finally, c) the *Crypto Service Manager* (CSM) [6], which provides software components with cryptographic functions implemented in software or hardware.

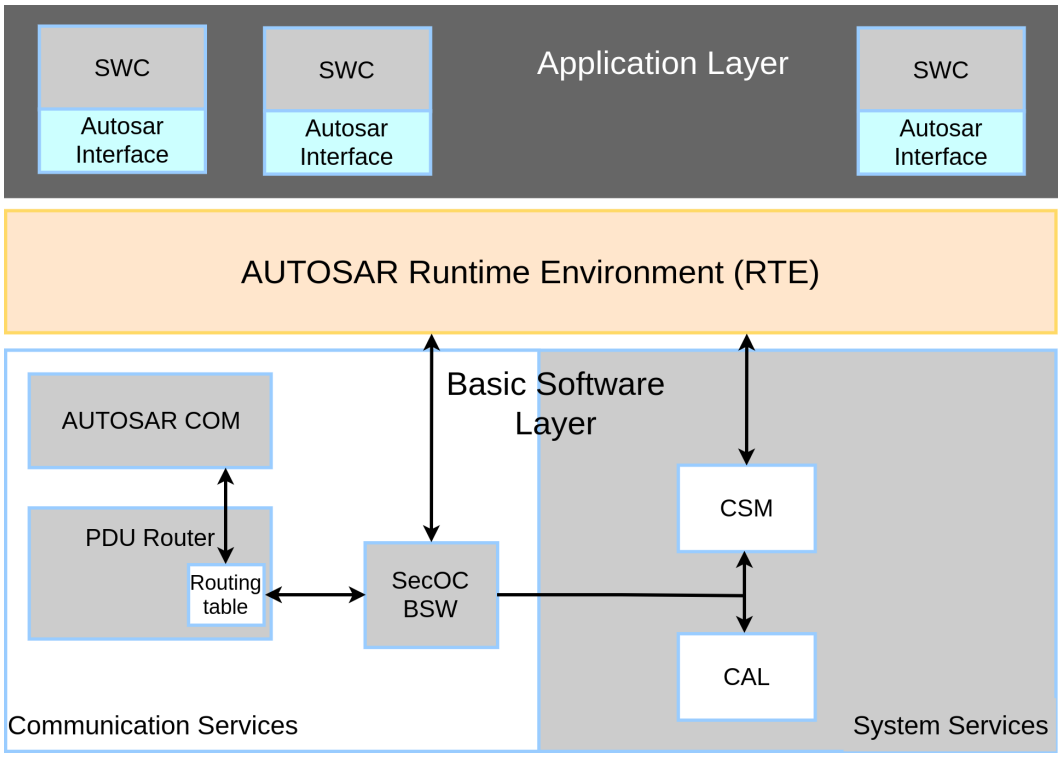

Fig. 1 AUTOSAR architecture.

The SWCs in the application layer communicate using ports that express client-server relationships or sender-receiver data interactions. The development of the SWCs relies on the RTE specified by AUTOSAR to deliver the conceptual foundation for the communication of SWCs with each other and the use of BSW services. The internal behaviour of SWCs consists of runnables or functional units, represented by a function entry point. Each runnable indicates the port it uses. Runnables internal to a SWC can communicate also through global variables and inter-runnable variables.

An example is shown in Figure 2. `Runnable1a` of SWC1 communicates with `Runnable2a` of SWC2 through sender-receiver ports; `Runnable1c` of SWC1 communicates with`Runnable1a` of the same SWC through an inter-runnable variable. Moreover, `Runnable2c` of SWC2 communicates with `Runnable1d` of SWC1 through a client-server port.

In [13], AUTOSAR models are extended with *security annotations*. In short, two modelling extensions are introduced:

- the *trust level* of a software component, or of a port
- the *security requirement* of a communication link

A software component (or a port) may be associated with a trust level which specifies to what extent it can be trusted to provide the expected function, or service, with respect to attacks targeting the component itself. Ports inherits the trust level of the SWC. Without loss of generality, we assume two trust levels: *high* and *low*. A communication link may be associated with a security requirement which represents the level of security that data sent on the link must satisfy. The security requirement can take one of the following values: *none*, *conf*, *integr*, *both*, which, respectively, codify no security, confidentiality, integrity and, both confidentiality and integrity. During the design phase of the automotive system, designers can assign these annotations to components and links according to their knowledge of the system.

As an example, let us consider the annotated AUTOSAR model shown in Figure 3. The example represents a typical active safety application that makes use of information coming from sensory input devices (e.g., lidars, radars, cameras, and GPS) in order to sense the surrounding environment and detect road marks and objects (e.g., vehicles, pedestrians) on and around the street. These information items are forwarded to several navigation and active safety functions, including, for example, Path planning, Lane keeping and Lane Departure warning, which produce commands for the actuation systems (steering, throttle and brakes).

The PathPlanning software component and port `p` of the Throttle software component are assigned *high* trust level, while the other elements are assigned *low*.

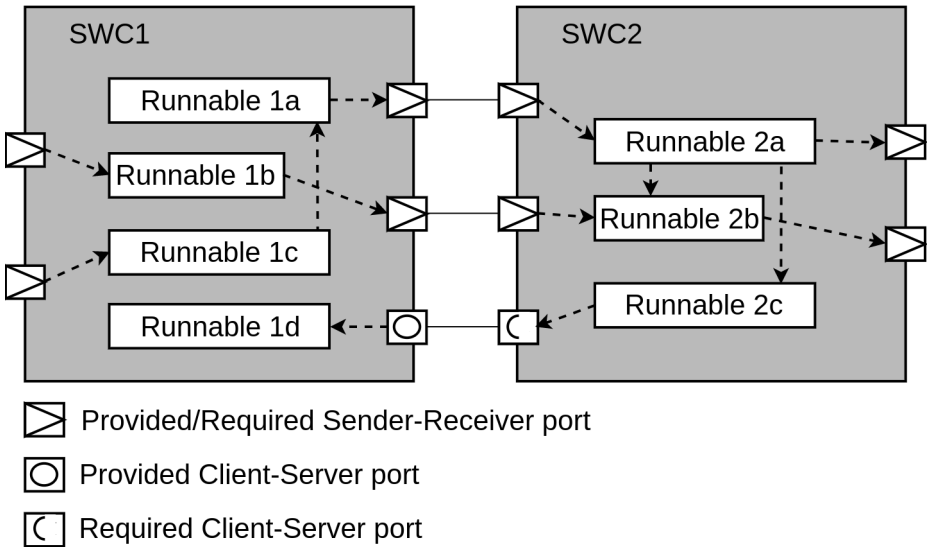

Fig. 2 An example of AUTOSAR application model.

The Throttle request link is annotated with integrity security requirement (*integr*), while the other communication links have no security requirements (*none*). Therefore, according to annotations, data input to the Throttle component at port `p`, on the Throttle_request link, must have a *high* trust level and integrity security requirement (*integr*).

3.2 Secure information flow

In this section we briefly recall basic concepts of secure information flow in a program [17].

A program, with variables partitioned into two disjoint sets of high and low security, has secure information flow if observations of the final value of the low security variables do not reveal any information about the initial values of the high security ones.

Assume `y` is a high security variable and `x` a low security one. Examples of violation of secure information flow are:

- `x := y;`
- `if (y = 0) then x := 0; else x := 1;`

In both cases, checking the final value of the low security variable `x` reveals information on the value of the higher security variable `y`. In the first case, there is an explicit information flow from `y` to `x` (variable `x` is assigned the value of `y`). In the second case there is an implicit information flow from `y` to `x`, since variable `x` is assigned different values depending on the value of the condition of the control instruction `if`, that depends on variable `y`.

A conditional instruction in a program causes the beginning of an implicit flow. The implicit flow begins when the conditional instruction starts (we say that we have an opened implicit flow); all the instructions in the scope of the `if` depends on the level of the condition of the `if`. In case of nested conditional instructions, we have the dependency from all the conditions of the opened implicit flows.

Information flow occurs also through global variables and function calls in the program. Finally, when a function call is executed in the scope of a conditional instruction, the function is executed under the implicit flow. For example,

- `if (y < 0) then f();`

Function `f()` is invoked depending on the value of variable `y`. Instructions in the code of `f()` are executed only if the value of variable `y` is less than `0`. Instructions of `f()` are executed under the implicit flow of the condition of the `if` statement.

The analysis of secure information flow can be executed using an abstract interpretation approach [16] based on the operational semantics of the language [10]. In this case

- the standard operational semantics of the programming language is enhanced to include information on security level of values.
- abstract domains are identified and abstract semantics rules are defined that execute the program on abstract domains that contain only security levels.
- the abstract rules compute the flow of information in the program.

In the following, the basic concepts of the analysis are shown. A program is a sequence of instructions $q = q_0 q_1 \cdots q_n$. Let m be a memory that contains all the

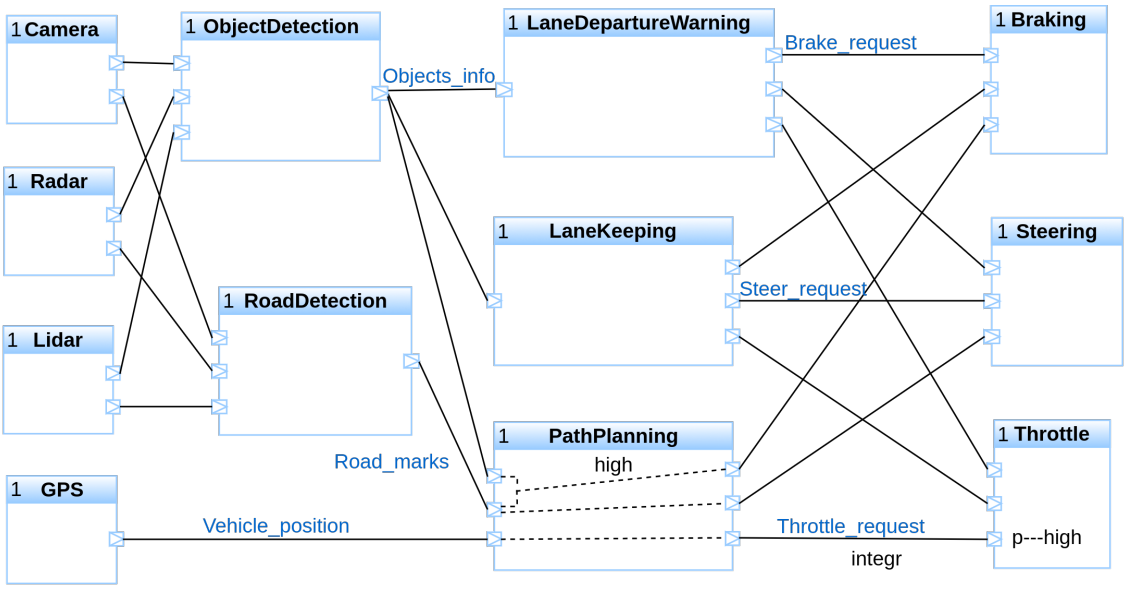

Fig. 3 An example of security annotated model in Rhapsody.

variables accessed by the program. The execution of the program is a transition system obtained by executing q starting from the initial memory m, by applying the rules of the operational semantics of the language.

The semantics is expressed by inference rules in the form $\frac{A}{C}$ where A is the antecedent and C is the consequent. The intuitive interpretation of a rule is that the consequent can be inferred from the antecedent.

Given a pair $\langle q_i, m\rangle$ of an instruction q_i and a memory m, $\longrightarrow$ represents the execution of q_i in m. The rule for a simple expression consisting of a variable `x` is:

$$\textbf{Expr}\quad \frac{}{\langle x, m\rangle \longrightarrow m(x)}$$

An empty antecedent corresponds to the boolean value `true`. It is always true (antecedent) that the evaluation of the expression `x` is the value of `x` in m (consequent). The rule for the assignment is :

$$\textbf{Ass}\quad \frac{\langle e, m\rangle \longrightarrow k}{\langle x := e, m\rangle \longrightarrow m[k/x]}$$

If the evaluation of the expression e in memory m is k (antecedent), then the execution of $x{:=}e$ changes memory m, by assigning value k to variable x (consequent). We assume $m[k/x]$ is a memory equal to m except for the variable x that is assigned k.

The operational semantics of the language is extended to convey the security level of data during the execution. We added two elements to the execution:

- *annotated values*. Each value is annotated with a security level τ, which considers the security level of all data on which the value depends.
  Data become pairs (k, τ), where k is the value and τ is the security level.
- *execution environment*. Each instruction q_i is executed under an environment σ that represents the level of the implicit flows caused by conditional instructions. For example, the level of a variable is given by the highest level between the level of the data in the variable and the level of the environment in which the instruction is executed.

We use the pair $\langle \hat{m}, Env\rangle$ to represent the memory defined on extended values ($\hat{m}$) and the execution environment (Env). The inference rules above become the following:

$$\textbf{Expr}\quad \frac{\hat{m}(x) = (k, \tau)}{\langle x, (\hat{m}, \sigma)\rangle \longrightarrow (k, \sigma \cup \tau)}$$

The notation $\langle x, (\hat{m}, \sigma)\rangle$ represents the evaluation of variable x in memory $\hat{m}$ under the environmnt σ.

$$\textbf{Ass}\quad \frac{\langle e, (\hat{m}, \sigma)\rangle \longrightarrow (k, \tau')}{\langle x{:=}e, (\hat{m}, \sigma)\rangle \longrightarrow \hat{m}[(k, \tau')/x]}$$

The notation $\langle e, (\hat{m}, \sigma)\rangle$ represents the evaluation of expression e in memory $\hat{m}$ under the environmnt σ.

If the level of the expression e is (k, τ') (antecedent), then the assignment updates the memory $\hat{m}$ assigning (k, τ') to variable x (consequent).

The abstract semantics abstracts from actual values and maintains only dependency levels. Let M be the abstract memory.

The inference rules above become:

$$\textbf{Expr} \quad \frac{M(x) = \tau}{\langle x, (M, \sigma)\rangle \longrightarrow \sigma \cup \tau}$$

$$\textbf{Ass} \quad \frac{\langle e, (M, \sigma)\rangle \longrightarrow \tau'}{\langle x{:=}e, (M, \sigma)\rangle \longrightarrow M[\tau'/x]}$$

A program is executed on the abstract domain starting from the abstract initial memory, and applying the abstract rules. In the abstract execution, all branches of conditional/iterative instructions are always executed, due to the loss of real data in the abstract semantics. Then the execution of the program with the abstract semantics captures all information flows.

4 Proposed approach

Given an annotated model, the main steps of the proposed approach are:

1. the computation of data dependencies;
2. the verification of data secure flow property;
3. the possible improvement of annotations.

We first give an overview of the proposed approach in subsection 4.1. Then, for the sake of simplicity, we introduce steps 2 and 3 in subsection 4.2 and, finally, we describe step 1 in subsection 4.3.

4.1 Overview

In the analysis, data are assigned a pair

$$\langle \texttt{trust level}, \texttt{security requirement}\rangle$$

that characterises their degree of security during all the possible executions. The initial values of `trust level` and `security requirement` are the most secure trust level and the most secure security requirement, respectively. As data flow through the components and the communication links, their degree of security is downgraded with the less secure annotation encountered.

Given an AUTOSAR model, data secure flow is verified if the degree of security of data sent on a link has no lower `trust level` and no lower `security requirement` than those assigned by the designer through the security annotations.

In particular, an AUTOSAR model satisfies data secure flow if for each link $l = (p_i, p_j)$

- the trust level of data sent on link l is not lower than the security annotation of port p_j;
- the security requirement of data sent on link l is not lower than the security annotation of link l.

Let us consider the AUTOSAR model shown in Figure 3. The model is correct if data sent on the Throttle_request link have a trust level greater than or equal to *high*, because the port `p` on the Throttle component has been assigned *high*.

Let us assume every component consists of a single runnable. Using the dependencies available in AUTOSAR, data sent by PathPlanning depend on all the inputs of PathPlanning. Therefore the trust level of data sent on the Throttle_request link is the lowest level of the traversed components (Camera, Lidar, Radar, GPS, etc.), *low* in this case. (By default, if not explicitly assigned, the `trust level` of components and the `security requirement` of links is the lowest level (`low` and `none`, respectively)).

The model is not correct, and to satisfy secure flow, all these components must be assigned *high*. However, this set of *high* trust level components can be oversized, because it does not rely on real dependencies. Let's consider the case in which real dependencies for output data of PathPlanning are known (dotted lines internal to the component in the figure). Since data sent on the Throttle_request ultimately depend only on data produced by GPS, secure flow is verified by simply assigning a *high* trust level to GPS. The resulting set of *high* trust level components is smaller than the previous one, thus reducing the overhead caused by security operations.

Similar reasoning applies to links. With reference to Figure 3, data on the Throttle_request link must have integrity security requirement (*integr*). Only knowing the ports each runnable reads and writes, data sent on the Throttle_request link depend on data at the input ports of the component PathPlanning, thus in order to satisfy secure flow all the involved links must guarantee integrity. Using the real dependencies, it is sufficient that the Vehicle_position communication link is assigned *integr* security requirement.

To have exact information on data dependencies requires knowing the code of runnables, and runnables must be executed on every possible input resulting in high complexity. This work proposes a solution based on a trade-off between the precision of the analysis and its complexity. An approximation of the real data dependency is computed using an abstract interpretation approach, which statically computes dependencies by abstracting from real values and considering only dependency levels. As a consequence, the set of causal dependencies found can be still oversized with respect to the exact dependencies, but it is more accurate with respect to the set obtained using the AUTOSAR information.

4.2 Data secure flow property

Given an AUTOSAR model, we use the following notations and definitions:

- $C = \{c_1, c_2, \cdots, c_k\}$ is the set of SWCs.
- R is the set of all runnables.
- VIR is the set of inter-runnable variables, VG is the set of global variables of SWCs.
- $P = \{p_1, \cdots, p_n\}$ is the set of ports of SWCs.
- $L = \{l_1, \cdots, l_m\}$ is the set of links. A link denotes a connection between two ports. Link $l = (p_i, p_j)$ connects port p_i to port p_j, with p_i output port of the sender SWC and p_j input port of the receiver SWC.
- $\texttt{cmp}(p)$ is the component to which port p belongs.
- $\texttt{trustlevel}(c)$ is the trust level assigned to software component c.
- $\texttt{trustlevel}(c, p)$ is the trust level assigned to port p of software component c.
- $\texttt{securityreq}(l)$ is the security requirement assigned to link l.
- $\texttt{Deps}: P \rightarrow 2^P$ is the function that provides the dependencies of ports. $\texttt{Deps}(p)$ is the set of ports on which the data written onto port p depend.

Let us introduce the following definitions.

Definition 1 (lattice) *Let A be a set and $\sqsubset$ an order relation on A. The pair $(A, \sqsubset)$ is a lattice if every pair of elements in A has both a greatest lower bound (*glb*) and a least upper bound (*lub*).*

Definition 2 (trust level) *Let $A = \{low, high\}$ be the set of trust levels, ordered by $low \sqsubset high$, where $\sqsubset$ is the lower between levels. $(A, \sqsubset)$ is a lattice. We have that $glb(low, high) = low$ and $lub(low, high) = high$.*

Definition 3 (security requirement) *Let $B = \{conf, integr, both, none\}$ be the set of security requirements of links, partially ordered by the $\sqsubset$, with $none \sqsubset conf \sqsubset both$ and $none \sqsubset integr \sqsubset both$. $(B, \sqsubset)$ is a lattice. We have that* conf *and* integr *are not ordered with respect to each other, because one is not "lower in security degree" than the other, $glb(integr, conf) = none$ and $lub(integr, conf) = both$.*

Definition 4 (Data secure flow property) *Given an AUTOSAR model with security annotations, the model satisfies the data secure flow property if for each link $l = (p_i, p_j) \in L$:*

$$\delta_l \not\sqsubset \texttt{trustlevel}(c, p_j) \wedge \mu_l \not\sqsubset \texttt{securityreq}(l)$$

where $c = cmp(p_j)$ and, δ_l and μ_l are the lowest trust level and the lowest security requirement of data sent onto link l.

In the analysis, we compute δ_l and μ_l with the algorithm shown in Listing 1. The output of the algorithm is highly dependent on function *Deps*.

Given a link $l = (p_i, p_j) \in L$,

1. $\langle \delta_l, \mu_l \rangle = \langle high, both \rangle$;
2. $\forall p \in Deps(p_i)$:
   $\delta_l = glb(\delta_l, \texttt{trustlevel}(cmp(p)))$;
3. $\forall l' = (q, q') \mid q, q' \in Deps(p)$:
   $\mu_l = glb(\mu_l, \texttt{securityreq}(l'))$.

Listing 1 Algorithm for data security of link l.

Given a link $l = (p_i, p_j) \in L$,

1. $\langle \delta_l, \mu_l \rangle$ are computed by Listing 1;
2. if $\delta_l \sqsubset \texttt{trustlevel}(cmp(p_j))$ then
   $\forall p \in Deps(p_j)$:
   $\texttt{trustlevel}(cmp(p)) =$
   $lub(\texttt{trustlevel}(cmp(p_j)),$
   $\texttt{trustlevel}(cmp(p)))$;
3. if $\mu_l \sqsubset \texttt{securityreq}(l)$ then
   $\forall l' = (p'_i, p'_j) \mid p'_i, p'_j \in Deps(p_j)$:
   $\texttt{securityreq}(l') =$
   $lub(\texttt{securityreq}(l), \texttt{securityreq}(l'))$.

Listing 2 Algorithm for updating the security annotations.

Let's assume $l = (p_i, p_j)$. Data sent to the link are data written onto port p_i. First the algorithm sets δ_l equal to the greatest trust level and μ_l equal to the greatest security requirement. Then for each port p on which data sent on the link l depends ($p \in Deps(p_i)$), δ_l is updated to consider the trust level of the port p: the trust level δ_l is set equal to the greatest lower bound between the current value and the trust level of the SWC to which port p belongs. Finally, for each link l' traversed by data sent on link l (source and destination ports of l' belong to $Deps(p_i)$), μ_l is set equal to the greatest lower bound between the current value and the security requirement of the link l'. Note that, at each step δ_l and μ_l can only be downgraded. Listing 1 is applied to each link within the AUTOSAR annotated model.

In order to satisfy the data secure flow property it is possible to update the annotated model as shown in Listing 2. Every link is analysed. For each link $l = (p_i, p_j)$ that violates the property ($\delta_l \sqsubset \texttt{trustlevel}(cmp(p_j))$ or $\mu_l \sqsubset \texttt{securityreq}(l)$) it is sufficient to accordingly increase the *trust level* of components whose ports belongs to $Deps(p_j)$ (point 2 in the listing) and the *security requirement* of the links

whose source and destination ports belongs to $Deps(p_j)$ (point 3 in the listing).

The fulfilment of data secure flow property is highly dependent on function *Deps*. For example by choosing an approach based on the information provided by AUTOSAR, we overestimate the set of dependencies, leading to the need for more security operations. By using an approach that analyses all the possible extecutions of runnables on the real input data we can implement a *Deps* function that retrieves the minimal set of dependencies. Our approach is set at an intermediate level of accuracy between the two.

4.3 Dependencies between data in AUTOSAR

In the previous section we defined the *data secure flow property* and how to use it to improve an annotated model, in the following we define *Deps* used in our approach.

Let us consider an AUTOSAR model. Data written at port p_j does not depend on data read from port p_i (p_j does not depend on p_i for short) if, changing the data at p_i, the data written onto p_j are always the same. We formally define port dependencies as follows.

Definition 5 (Port dependencies) *Given a model, let $p_j(p_1, \cdots, p_{i-1}, v, p_{i+1}, \cdots, p_n)$ be the data written onto port p_j when v is read from input port p_i. A port p_j does not depend on the port p_i if:*
for each possible execution, for each pair of data v_1, v_2 at p_i, with $v_1 \neq v_2$, it is:

$$p_j(p_1, \cdots, p_{i-1}, v_1, p_{i+1}, \cdots, p_n) = p_j(p_1, \cdots, p_{i-1}, v_2, p_{i+1}, \cdots, p_n)$$

Dependencies between data are computed by applying an abstract interpretation approach, similar to the one described in Section 3. The difference is that in our work the abstract domain consists of port levels instead of security levels.

4.3.1 Dependency levels.

In the analysis we define the set of data dependency levels Σ as the power-set of P: $\Sigma = 2^P$, i.e. the set of all subsets of P, ordered by subset inclusion. The set Σ with the ordering relation $\subseteq$ is a lattice $(\Sigma, \subseteq)$ (i.e., every pair of elements of Σ has both a greatest lower bound, *glb*, and a least upper bound, *lub*). The *lub* is given by the union ($\cup$) and the *glb* is given by the intersection of subsets ($\cap$). Given $X \subseteq Y$, $X \cup Y = Y$ and $X \cap Y = X$. The singleton set $\{p_i\}$, (p_i for short) denotes a dependency from port p_i. The set $\{p_i, p_j\}$ denotes dependency on both ports p_i and p_j. The minimum of Σ is the empty set $\emptyset$, the maximum is $\{p_1, p_2, \cdots, p_n\}$($P$ for short).

We extend an AUTOSAR model with the lattice of dependency levels $(\Sigma, \subseteq)$.

4.3.2 Analysis of an AUTOSAR model

Let us now consider the analysis of an AUTOSAR model. The basic idea consists in modelling ports as variables, and runnables as functions.

In particular,

- for sender-receiver data communications, reading data from a port is equivalent to reading a variable; writing data onto a port is equivalent to writing a variable.
- for client-server communications, the client request is equivalent to a function call, which corresponds to the invocation of the runnable implementing the requested service.

Runnables are functions, with arguments (passed by value or by reference) and return. In addition to a local memory, runnables have access to a global memory that maintains inter-runnable variables, global variables of SWCs, and communication ports. We call the set of these elements global context. In particular, in the analysis, runnables are executed in a global context A and in a local context $\langle M, Env \rangle$, which consists of a local memory M and an execution environment Env, see Subsection 2.3.

Since runnables are Misra-C compliant [3], we need to deal with pointers, structures and arrays. Misra-C [28] is a subset of C language, specifically addressed for safety-critical systems.

In particular:

- a pointer is assumed to be a simple variable, that maintains the dependencies of the pointer, plus the dependencies of the pointed data in the abstract execution.
- a structured variable is mapped to a set of simple variables, one for each member (we use the $\cdot$ notation, as usual). If we have a variable *data* that is a structure with two fields a and b, we map such a variable into two simple variables, named *data.a* and *data.b*, respectively.
- An array is assumed to be a simple variable, that maintains all the dependencies of each element in the array.

The analysis of an AUTOSAR model is based on an iterative process that performs the abstract execution of all runnables in R, using the global context file. If during the analysis a level in the global context file changes, all runnables must be re-executed.

$A := A^{\emptyset}$
$T := R$
`while`$(T \neq \emptyset)$
  `select` $r \in T$
  $T := T - \{r\}$
  $A' := EXEC(r, A)$
  `if`$(A' \neq A)$
    $A := A'$
    $T := R$

Listing 3 Analysis of an AUTOSAR model

The main steps of the iterative analysis are shown in Listing 3, where A^0 is the initial global context file.

The analysis uses the abstract interpreter EXEC to analyse a single runnable. EXEC performs an abstract execution of the runnable starting from a global context file A and producing a new global context file A'.

The analysis terminates when, starting from a global context file, all runnables are executed and the global context is not changed.

At the end of the analysis the global context file records the dependencies for ports of all the SWCs. The approach is conservative, in the sense that all possible dependencies for any real execution of the runnables are detected. False dependencies are possible, since, for example, in the abstract analysis all branches of control instructions are executed, even those that in real execution would never have been executed.

Note that our approach analyses runnables independently of one another. The analysis does not require the explicit construction of the complete call graph of runnables that would generally result in a large number of states. Moreover, the approach scales well in terms of computational time with the size of the system because all the runnables can be analysed in parallel.

5 Implementation

This section provides the practical methods used to implement the analysis depicted in the previous section. We focus on the resolution of RTE calls, global and local contexts management and abstract execution of runnables.This section also provides an example to better clarify the analysis. Finally this section provides details on the architecture of the tool implemented.

5.1 Calls to RTE functions

In the following we show how to deal with calls to RTE functions in the runnable code, through a few exemplary cases.

- Data communication ports
  RTE functions for reading from or writing onto ports are mapped so as to read from and write onto the port variable. For the sake of simplicity, the name of the port variable is the same name as the port.

  The `ReturnType Rte_Write_Port_o(data)` function, where `data` is the function's argument and `o` is the port, is implemented as the assignment `o = data`.

  The `ReturnType Rte_Read_Port_o()` function, which returns the data read from port `o`, is implemented as the expression `o`.

- Service ports
  RTE functions that invoke remote services trigger the runnable that implements the service. The function implementing the service is invoked.

  The function `ReturnType Rte_Call_Port_o(data)`, where `o` is the service (runnable) within the client-server interface and `data` are the arguments of `o`, is implemented as `o(data)`.

5.2 Global Context and local context

The global context records information on variables in the global memory of SWCs, communication ports of SWCs and runnable calls.

The global context file maintains:

- for each variable $v \in IRV \cup GV \cup P$, the entry $v : \tau$, where $\tau \in \Sigma$ is the dependency level of v;
- for each runnable $r \in R$, the entry $r(\tau_1, ..., \tau_k)\tau; \sigma$, where $\tau_1, ..., \tau_k$ are the levels for the actual parameter, τ is the level for the return and σ is the level of the environment under which the runnable is executed (calling environment).

During the analysis, for each variable, the global context maintains the maximum dependency level of data recorded in the variable and, for each runnable, the global context maintains the maximum level of the arguments, the maximum level of the return and the maximum level of the environment, by considering all the possible invocations.
On the other hand, the local context of the runnable is the pair (M, Env). The local memory M contains local variables (including variables modelling arguments passed by value), and the environment Env which is the level of the implicit flow caused by conditional instructions. The return of a runnable and runnable's ar-

guments passed by reference are handled as global variables.

At the beginning of the analysis, the global context A is initialised as follows:

- each port variable p_i depends only on itself, and so it is initialised to the level $\{p_i\} \in \Sigma$.
- all other variables are initially assigned to the lowest level ($\emptyset$).
- runnables are initially assigned $\emptyset$ for calling environment, parameters and return.

For the local context, the local memory M is initialised as follows:

- local variables are initialised to $\emptyset$.
- variables corresponding to arguments passed by values are initialised to the level of the argument in the local context.

The execution environment Env is initialised with the level of the calling environment of the runnable entry in the global context file.

Listing 4 shows an example of the general structure of a global context file. In the example, runnable `run1()` has one argument passed by value, and one argument passed by reference (denoted by `arg&` hereinafter).

The global context is used to take into account the interactions between runnables. Any update to the global context is permanent, and visible to other runnables. The local context of a runnable is deallocated when the analysis of the runnable terminates.

5.3 Abstract execution of a runnable

A runnable is executed by an abstract interpreter EXEC which takes as input the CFG of the runnable. In the CFG the instructions are grouped in Basic Blocks (`bb`).

Each type of instruction is assigned an abstract execution rule. The abstract execution of an instruction, updates the local context (M, Env), and the global context A. EXEC examines one `bb` at a time and abstractly executes each instruction of the block, and propagates the updates ((M, Env) and A obtained after the execution of the instructions) to successor blocks. Instructions in the scope of the conditional block, are executed under the implicit flow of the condition of the control instruction in the conditional block. The set of abstract rules is shown in Appendix A.

Examples of CFGs are shown in Figure 4 and Figure 5. In the first case there is an `if` instruction, while the second shows an example of a `while` instruction (see Listing 5).

```
% Begin Global Context
% global variables of SWCs
gv1 = {}
gv2 = {}
.....

% IRV of SWCs
irv1 = {}
irv2 = {}
.....

% ports of SWCs
p1 = {p1}
p2 = {p2}
....

% runnables of SWCs
run(a,{}) {}; {}
run1((b, {}), (c&, {})) {}; {}
run2() {}; {}

.....
%End Global Context
```

Listing 4 An example of initial global context.

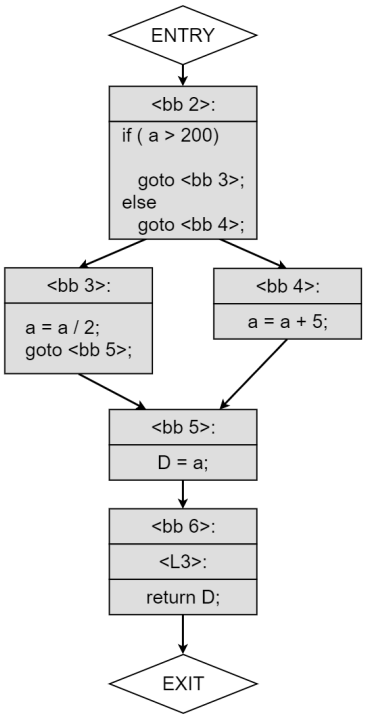

Fig. 4 CFG of runnable1 with `if`.

In Figure 4 blocks in the scope of the `if` statement (block 2), are blocks 3 and 4 (which are the successors of block 2).

In Figure 5, we note that the `while` statement is translated into a repeated `if` instruction (block 4). In this case only one of the successors of the conditional

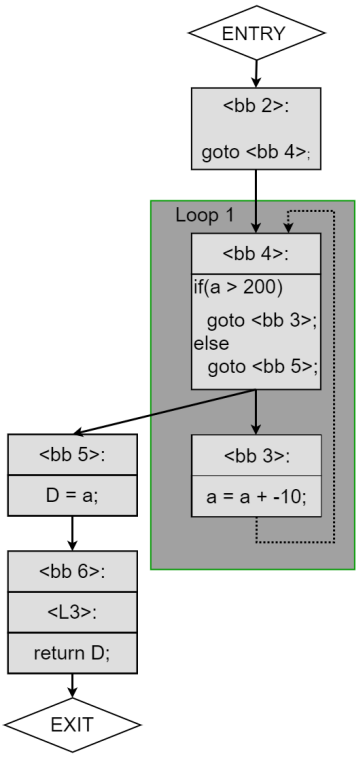

Fig. 5 CFG of runnable2 with `while`.

block (block 3) is in the scope of the `if`.

```
int runnable1 (int a) {
   if (a>200)   a = a/2;
     else   a= a+5;
   return a;
}
int runnable2 (int a) {
   while (a>200)
        a = a -10;
   return a;
}
```

Listing 5 Code of runnables.

EXEC iteratively performs the abstract execution of the runnable starting from an initial local context $\langle M, Env\rangle$ and an initial global context A until a fix-point is reached (i.e., during an iteration $\langle M, Env\rangle$ and A are updated, and the analysis terminates when the local and global context at the beginning and at the end of the iteration are the same). The order in which blocks are executed is not important, because if the A or $\langle M, Env\rangle$ change, all blocks are re-executed.

EXEC uses a table Q that implements the local context of a runnable (local memory M plus calling environment Env). Q consists of a row Q_i for each `bb` i in the CFG.

If $Q_i = \langle M, \sigma\rangle$, we have that M contains the level of the local variables when block i is executed, and σ is the level of the environment in which block i is executed. Q_i is named before-state of block i.

The execution of the instructions of block i starts from the before-state of i. The execution of the instructions generates a new state $\langle M', Env'\rangle$, named after-state of block i. The after-state is obtained as the result of the abstract execution of each instruction, according to the abstract rules in Appendix A.

After the execution of all instructions of block i, the content of the memory M' in the after-state is propagated to the successors of i. For each successor j, the before-state of j (row Q_j in the table) is updated by executing the *lub* operation between the memory of the after-state of block i with the memory of Q_j.

We naturally extend the *lub* operation to memories. This corresponds to the least upper bound executed point-wise on each variable in the memory:
$lub(M, M') : \forall var, M(var) = M(var) \cup M'(var)$.
Q_{entry} and Q_{exit} represent the entry and the exit block of the runnable, respectively.

Table T summarises information on `bb`s of the runnable. The column *Code* contains the code of the `bb`, the column *Succ.* enumerates the successors of the `bb` and the column *Scope* reports the blocks in scope of `bb`. Table 1 shows T for runnable2 in Figure 5.

The set of abstract rules is shown in Appendix A.

5.4 An example

Let's consider runnable2 in Figure 5. Assume we have the following fragment of global context:

```
% Begin Global Context
....
runnable2((a, {p2})) 0 : {p1}
...
% End  Global Context
```

The initial local context Q is shown in Table 2. The environment of each row is initialised with the value of the calling environment in the global context file ($\{p1\}$). The memory of Q_{entry} is computed as follows: local variables are all assigned the minimum level ($\emptyset$), except the variable corresponding to argument a which assumes the value present in the global context file ($\{p2\}$). All the variables in the memory of other blocks are assigned the minimum level $\emptyset$.

In the following, M_{Q_i} is the memory of the before-state of Q_i and M'_{Q_i} denotes the memory of the after-state of Q_i.
Let blocks be scheduled in ascending order of i.

Table 1 Table T of runnable2.

Block	Code	Succ.	Scope
Q_{entry}		2	
Q_2	goto bb4;	4	
Q_3	a = a-10;	4	
Q_4	if (a > 200) goto bb3 else goto bb5	3, 5	3
Q_5	d = a	6	
Q_6	return d	exit	
Q_{exit}			

Table 2 Initial local context Q of runnable2.

Block	Memory		Env
Q_{entry}	$(a, p2)$	$(d, \emptyset)$	$p1$
Q_2	$(a, \emptyset)$	$(d, \emptyset)$	$p1$
Q_3	$(a, \emptyset)$	$(d, \emptyset)$	$p1$
Q_4	$(a, \emptyset)$	$(d, \emptyset)$	$p1$
Q_5	$(a, \emptyset)$	$(d, \emptyset)$	$p1$
Q_6	$(a, \emptyset)$	$(d, \emptyset)$	$p1$
Q_{exit}	$(a, \emptyset)$	$(d, \emptyset)$	$p1$

- Block `entry` simply initialises the memory of its successor (block 2) $M_{Q_2} = ((a, p2)(d, \emptyset))$.
- Block 2 is executed. The rule for `goto` does not change the memory, so $M'_{Q_2} = M_{Q_2}$. Block 2 propagates the after-state to its successor (block 4): $M_{Q_4} = lub(M_{Q_4}, M'_{Q_2}) = ((a, p2)(d, \emptyset))$.
- When block 3 is executed, the rule of the assignment is applied. The lub between the environment ($p1$) and the level of a in M (that is $\emptyset$) is computed and assigned to a. $M'_{Q_3} = ((a, \{p1\})(d, \emptyset))$. The memory of the successor blocks is updated: $M_{Q_4} = lub(M_{Q_4}, M'_{Q_3})$. Therefore, $M_{Q_4} = ((a, p2)(d, \emptyset)) \cup ((a, \{p1\})(d, \emptyset))$. That is, $M_{Q_4} = ((a, \{p1, p2\})(d, \emptyset))$.
- Block 4 is executed and the rule for `if` is applied. The level of the condition is $\{p1, p2\}$. The environment of the blocks in the scope of block 4 is updated, i.e., Env of block 3 becomes $\{p1, p2\}$. Then $M_{Q_3} = lub(M_{Q_3}, M'_{Q_4}) = ((a, \{p1, p2\})(d, \emptyset))$ and $M_{Q_5} = lub(M_{Q_5}, M'_{Q_4}) = ((a, \{p1, p2\})(d, \emptyset))$.
- The execution of block 5, assigns $\{p1, p2\}$ to d: $M'_{Q_5} = ((a, \{p1, p2\})(d, \{p1, p2\}))$ The after-state is propagated to the successor: $M_{Q_6} = lub(M_{Q_6}, M'_{Q_5}) =$ $((a, \{p1, p2\})(d, \{p1, p2\}))$.
- The execution of block 6, according to the rule for `return`, assigns $\{p1, p2\}$ to the return of the runnable in the global context file A and propagates the memory ($M'_{Q_6} = M_{Q_6}$) in the after-state to block `exit`.

Table 3 Local context Q of runnable2 after the first iteration (equal to the table at fixpoint).

Block	Memory		Env
Q_{entry}	$(a, p2)$	$(d, \emptyset)$	$p1$
Q_2	$(a, p2)$	$(d, \emptyset)$	$p1$
Q_3	$(a, \{p1, p2\})$	$(d, \emptyset)$	{p1, p2}
Q_4	$(a, \{p1, p2\})$	$(d, \emptyset)$	$p1$
Q_5	$(a, \{p1, p2\})$	$(d, \emptyset)$	$p1$
Q_6	$(a, \{p1, p2\})$	$(d, \{p1, p2\})$	$p1$
Q_{exit}	$(a, \{p1, p2\})$	$(d, \{p1, p2\})$	$p1$

$M_{Q_{exit}} = lub(M_{Q_{exit}}, M'_{Q_6})$. Therefore, $M_{Q_{exit}} = lub(((a, \emptyset)(d, \emptyset)), ((a, \{p1, p2\})(d, \{p1, p2\}))$ $= ((a, \{p1, p2\})(d, \{p1, p2\}))$.

When all the blocks have been analysed, the first iteration terminates. Local context Q obtained at the end of the first iteration is shown in Table 3.

The global context file, at the end of the iteration is the following:

```
% Begin Global Context
....

run((a, p2)) {p1, p2}   : p1
...
% End Global Context
```

Another iteration needs to be executed, because the context has been changed. At the end of the second iteration, since Table 3 does not change, the fixpoint is reached and EXEC terminates.

5.5 The tool

In the following we describe the architecture of the ADEPT tool (Autosar DEPencencies Tool) for computing data dependencies in AUTOSAR models.
The tool requires the CFG of the runnable entities and the global context structure to compute the ports dependencies. The global context structure contains the key information required by the tool for generating the initial global context (e.g. number of runnables, number of ports, number of inter-runnable variables) and this information can be automatically obtained with a simple scan of the code. The CFG of the runnables can be automatically extracted from the C code of the runnables using the GCC compiler with the following developer options: -fdump-tree-cfg-blocks-vops.
The tool has been developed entirely in C++ and consists of three main units: PARSER, RULES DB and ABSTRACT ENGINE (AE), see Figure 6. PARSER unit divides the CFG of runnables into subsequent tokens. AE is the core unit of the tool, it takes as input the rules of the analysis, stored in the RULES DB unit,

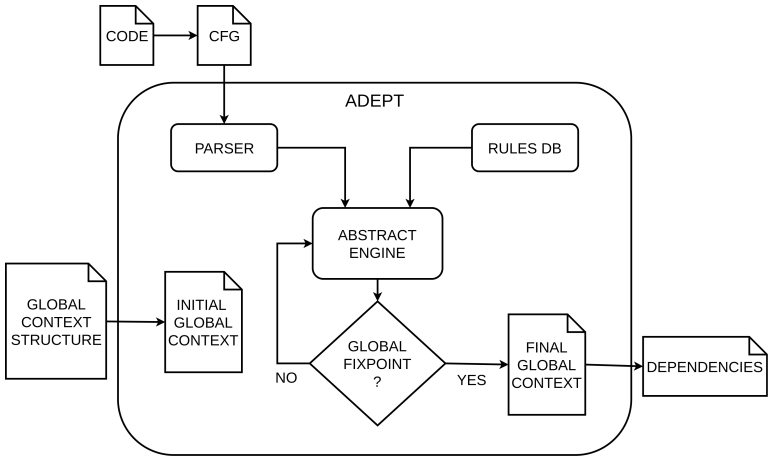

Fig. 6 Architecture of the tool.

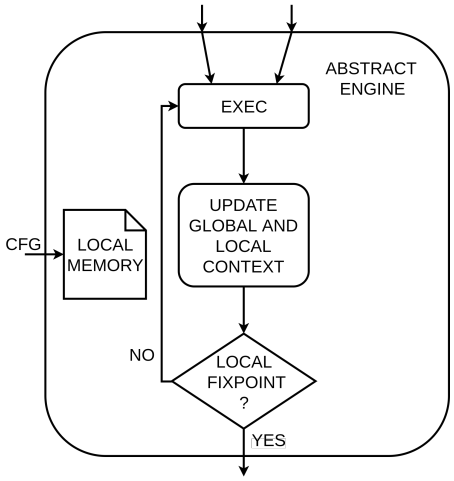

Fig. 7 Behaviour of AE

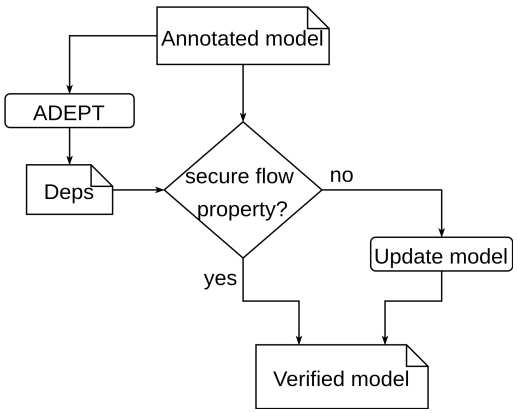

Fig. 8 Usage of the tool in the approach.

and the generated tokens, line by line and produces as output an update to the global context. Within AE, EXEC performs the abstract execution of the runnable by analysing the tokens and looking for predefined patterns. When one of these is found, EXEC adopts the proper rule and the local and global contexts are properly updated, see Figure 7.

Runnable entities are analysed one by one. The analysis of a runnable is iteratively executed and terminates when the local fixpoint is reached, i.e. when the local and global context do not change after an iteration. Once local fixpoint has been reached, AE moves on to analyse the next runnable. When all the runnables have been analysed, AE terminates, and the tool checks if the global fix point has been reached, i.e., the global context does not change after the execution of AE. If the global fixpoint is reached, the tool terminates, providing the final global context file with the port dependencies, otherwise AE is executed again.

Figure 8 shows how to use the tool in the proposed approach. Starting from an AUTOSAR model, ADEPT computes for each port p, the set of ports from which p depends ($Deps(p)$). Then, *data secure flow property* is checked using $Deps$ to find the lowest trust level and the lowest security requirement of data sent on every link. In case of violation, we change the annotations within the Rhapsody model according to the algorithm in Listing 2.

6 A case study

In the following, we will consider a use case related to the Front Light Manager (FLM) tutorial described in the standard documents of AUTOSAR [4], which is focused on a very limited functional part of the front light manager, namely activating the headlight and the daytime running lights. All other lights functions (e.g. parking orientation light, fog lights, etc.) are excluded.

In particular, we consider a slightly extended version of the FLM in which:

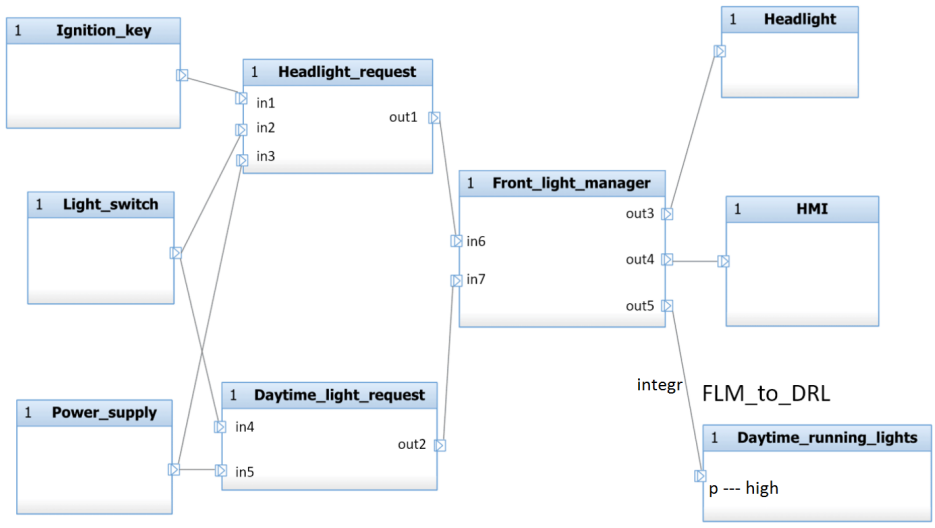

Fig. 9 Model of the Front Light Manager.

– the headlights are turned on if the key ignition is activated, the light switch is on and the power supply voltage is within a specific range,
– meanwhile the daytime running lights are turned on if the light switch is on and the voltage from the power supply is within a specific range.

The status of the lights is reported to the driver by means of the HMI (Human Machine Interface).

We created a model with a total of 9 components which can be divided in 3 categories:

– Sensors components: Ignition Key, Light Switch and Power Supply
– Actuator components: Headlight, HMI and Daytime Running Lights
– Control components: Headlight request, Daytime Light Request and Front Light Manager

The system model described using Rhapsody is shown in Figure 9. All the ports between these components are data Receiver-Provider ports. Ignition Key and Light Switch are assumed to be simply sensor components that outputs their status and Power Supply is assumed to simply outputs the battery voltage, therefore there is no need to further develop an implementation of them. In the following we will analyse the three control software components.

6.1 Control Software Components

The Headlight_request software component is made of three runnables entities, each with a different task:

– Runnable1 receives the data from Ignition Key and forwards it to Runnable3.
– Runnable2 receives the data from Light Switch and forwards it to Runnable3.
– Runnable3 receives data from Power Supply and from the other Runnables and sends a request of headlights activation to Front Light Manager.

The schema of the Headlight_request component is shown in Figure 10. Runnable1 and Runnable2 receive the data from input port `in1` and `in2` respectively, and forward their values to Runnable3 by means of two InterRunnable Variables, `IRV1` and `IRV2`. Runnable3 receives the value of the two IRVs and the voltage value from `in3` and checks if the voltage is within a specific range and both IRVs are 'ON'. If so it sends a request of headlights activation to Front Light Manager otherwise it stops sending request.

The Rte_IStatus_Runnable3_RPort_in3() is a function that returns the current status of the port in3, and '0' means 'no errors'. The voltage thresholds are assumed to be two global parameters of the system. The other functions are standard call to RTE functions used to read from or write onto ports or inter-runnable variables. The Daytime_light_request software component is made of only two runnables who act like Runnable2 and Runnable3 of the Headlight Request component. The Front_light_manager software component is made of three runnables:

– Runnable1 receives the request from the Headlight Request, and forwards it to the Runnable3.
– Runnable2 receives the request from the Daytime Light Request and forwards it to the Runnable3.
– Runnable3 forwards the data to the output ports and sends a signal to the HMI actuator if at least 1 of the two kinds of lights is requested on.

```
void HR_Runnable3(void) {
    int16_T input_voltage;
    // Check port status (0 --> no error)
    if (Rte_IStatus_Runnable3_RPort_in3() == 0){
        input_voltage = (int16_T)Rte_IRead_Runnable3_RPort_in3();
    }
    if ((input_voltage >= voltage_threshold1) &&
        (input_voltage <= voltage_threshold2)){
            if ((Rte_IrvIRead_Runnable3_IRV1() == KEY_ON) &&
               (Rte_IrvIRead_Runnable3_IRV2() == LIGHT_ON)){
                  Rte_IWrite_Runnable3_PPort_out1(REQ_HEADLIGHT_ON);
            }
            else{
                Rte_IWrite_Runnable3_PPort_out1(REQ_HEADLIGHT_OFF);
            }
    }
    else{
        Rte_IWrite_Runnable3_PPort_out1(REQ_HEADLIGHT_OFF);
    }
}

void FLM_Runnable3(void) {
    Rte_IWrite_Runnable3_PPort_out3(Rte_IrvIRead_Runnable3_IRV1());
    Rte_IWrite_Runnable3_PPort_out5((Rte_IrvIRead_Runnable3_IRV2());
    if ( (Rte_IrvIRead_Runnable3_IRV1() == REQ_HEADLIGHT_ON) ||
         (Rte_IrvIRead_Runnable3_IRV2() == REQ_DAYTIME_ON) ){
                Rte_IWrite_Runnable3_PPort_out4(LIGHTS_ON);
    }
    else{
          Rte_IWrite_Runnable3_PPort_out4(LIGHTS_OFF);
    }
}
```

Listing 6 Code of HR_Runnable3 and FLM_Runnable3.

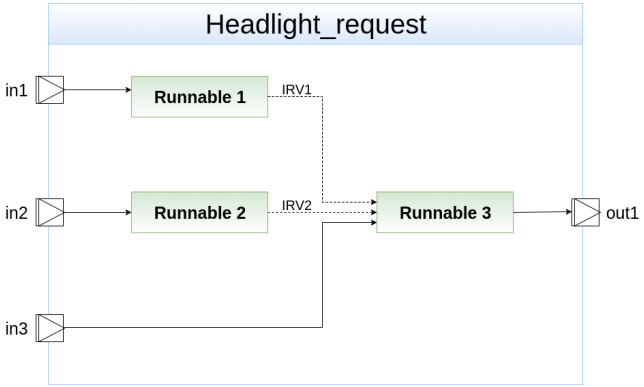

Fig. 10 Software component of Headlight Request.

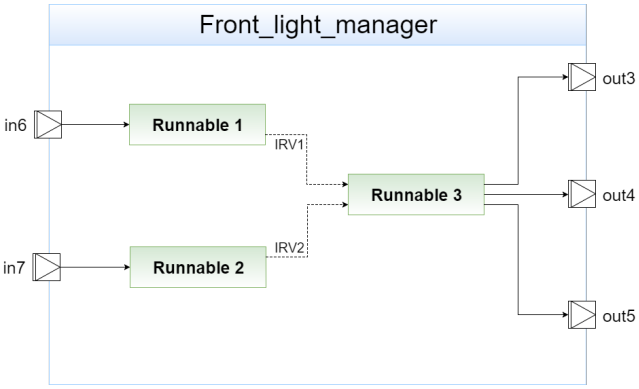

Fig. 11 Software component of Front Line Manager.

The schema of the component is shown in Figure 11. Runnable1 receives the data from input port `in6` and forwards it to inter-runnable variable `IRV1`. Runnable2 receives the data from input port `in7` and forwards it to inter-runnable variable `IRV2`. Runnable3 forwards the data stored in `IRV1` and `IRV2` to `out3` and `out5`, respectively, and if at least one of the request is 'ON' it sends the signal to turn the lights on to the `out4` port, otherwise it sends the signal to turn the lights off to the same port. The code of both FLM_Runnable3 and HR_Runnable3 is shown in Listing 6.

Since the basic example of the AUTOSAR standard considers the daytime lights as emergency lights to be used in the case of failure of the headlights, we assumed that a developer would request data input to Daytime_running_lights be generated by high trusted

software components, and the FLM_to_DRL link between the Front_light_manager and the software component Daytime_running_lights satisfies integrity requirement. This corresponds to the annotation shown in Figure 9, where the port `p` of Daytime_running_lights is assigned *high* trust level and the FLM_to_DRL link is assigned *integr* security requirement. All the other components and links are assigned *low* and *none*, respectively.

An excerpt of the global context structure file, input to the tool for the generation of the global context, is the following:

```
% global variables
int HR_voltage_threshold1;
int HR_voltage_threshold2;
int DLR_voltage_threshold1;
...
%  inter runnable variables
int16_t FLM_IRV1;
int16_t FLM_IRV2;
int16_t DLR_IRV1;
...
%  ports
int in1;
int in2;
...
int out1;
int out2;
...
% functions
void flm_Runnable1() 0;
void flm_Runnable2() 0;
.....
%  links
out2 -> in7;
out1 -> in6;
```

6.2 Tool application

Figure 12 reports the global context at the beginning of the analysis of the AUTOSAR model. The position (i, j) indicates if element i depends on port j. The boxed region of the matrix shows dependencies between ports. The analysis starts, assuming that each port depends only on itself (diagonal of boxed sub-matrix equal to 1).

Using a computer with Intel Core i7-4700MQ and 12 Gb of Ram the analysis completes in 349 *ms* and the global fixpoint has been reached after 3 iterations of AE.

Figure 13 reports the global context at the end of the analysis of the AUTOSAR model. For example, we derive that port `in6` depends on ports `in1`,`in2`,`in3`, `in6` and `out1`, so $Deps(in6) = \{in1, in2, in3, in6, out1\}$

In order to check *data secure flow* property, the algorithm in Listing 1 in Section 4 is applied. Let us consider link FLM_to_DRL. From Figure 13, we derive $Deps(out5) = \{in4, in5, in7, out2, out5\}$. Let L' be the set of links whose ports belong to $Deps(out5)$, it is $(out2, in7) \in L'$.

Step 1:
$\langle \delta_{FLM\_to\_DRL}, \mu_{FLM\_to\_DRL} \rangle = \langle high, both \rangle$.

Step 2: $\forall p \in Deps(out5)$:
$\delta_{FLM\_to\_DRL} = glb(\delta_{FLM\_to\_DRL}, \texttt{trustlevel}(cmp(p))$
Let us consider port $in4 \in Deps(out5)$.
It is $cmp(p) = Daytime\_light\_request$, whose trust level is *low*.
We have:
$\delta_{FLM\_to\_DRL} = glb(\delta_{FLM\_to\_DRL}, low) = low$
Since $\delta_{FLM\_to\_DRL} \sqsubset \texttt{trustlevel}(p)$, *data secure flow* is not satisfied.

Step 3: $\forall l \in L'$:
$\mu_{FLM\_to\_DRL} = glb(\mu_{FLM\_to\_DRL}, \texttt{securityreq}(l))$
Let us consider link $(out2, in7) \in L'$, whose security requirement is *none*.
We have:
$\mu_{FLM\_to\_DRL} = glb(\mu_{FLM\_to\_DRL}, none) = none$
Since $\mu_{FLM\_to\_DRL} \sqsubset \texttt{securityreq}(FLM\_to\_DRL)$, *data secure flow* is not satisfied.

Using the information of AUTOSAR we assign *high* trust level to all components directly or indirectly connected to Daytime_running_lights, which will lead to the assignment of *high* trust level to all the other components (Front_light_manager, Headlight_request, Daytime_light_request, Light_switch, Ignition_key, Power_supply).

With our approach we can consider the data dependencies of all the three components that we have implemented and we can exploit these dependencies in order to obtain a more efficient solution, in term of less overhead for security operations.
In particular, the output port of Front_light_manager connected to the Daytime_running_lights (`out5` in our implementation) does not depend on the input port connected to the Headlight_request component (`in6` in our implementation).

Data sent on the link `FLM_to_DRL` depends on Front_light_manager, Daytime_light_request, Light_switch and Power_supply software components and traverse the following links: `DLR_to_FLM`, `PS_to_DLR` and `LS_to_DLR`. As a consequence, *data*

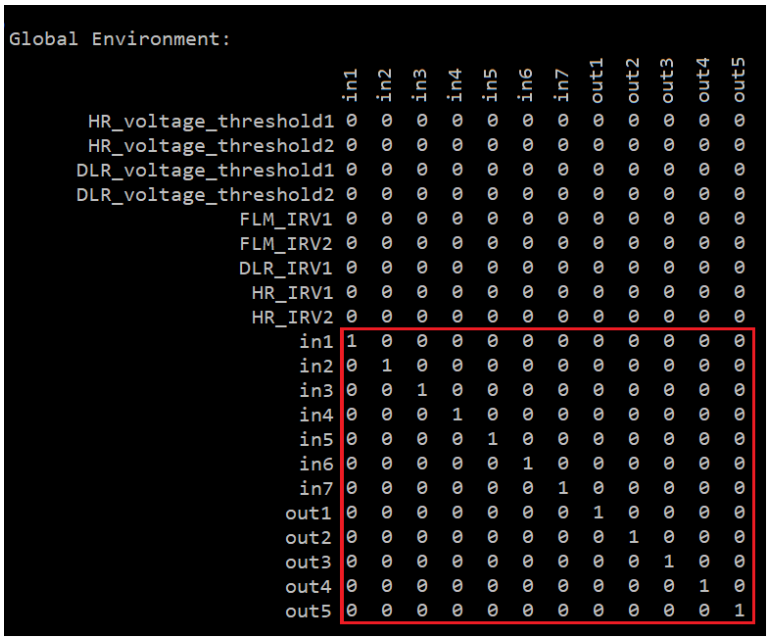

Fig. 12 Global Context file at the beginning of the analysis.

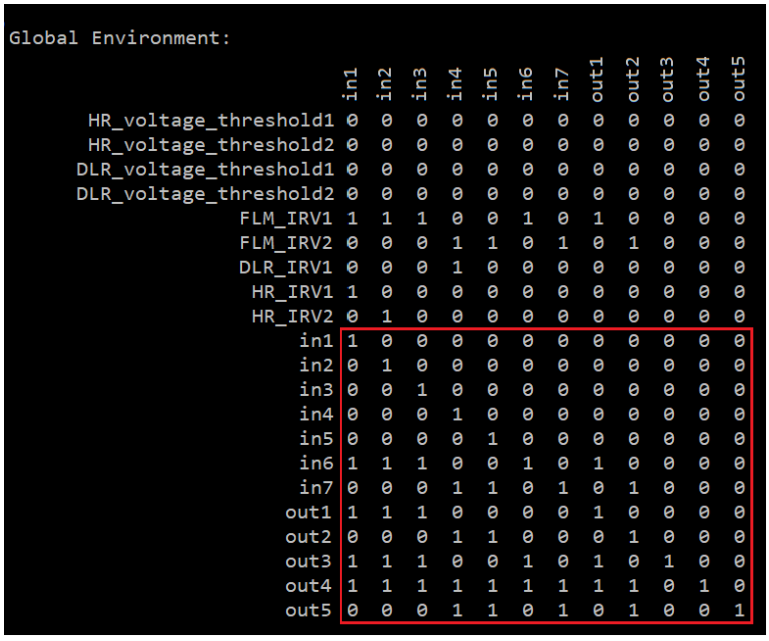

Fig. 13 Global Context file at the end of the analysis.

secure flow property requires that previous SWCs are assigned `high` trust level, and previous links are assigned `integr` security requirement. The resulting security annotated AUTOSAR model is shown in Figure 14.

7 Conclusions

Security in automotive systems is becoming increasingly important and should be taken into account from the early stages of the system design. There are a lot of well-known techniques and tools that can be borrowed from the information security domain in order to deal with malicious intrusions on automotive systems. In this paper *data secure flow* property has been defined, and an approach for the verification of such a property is presented. The approach is based on information flow analysis and abstract interpretation. The analysis computes the lowest security level of data sent on a communication, according to the annotations in

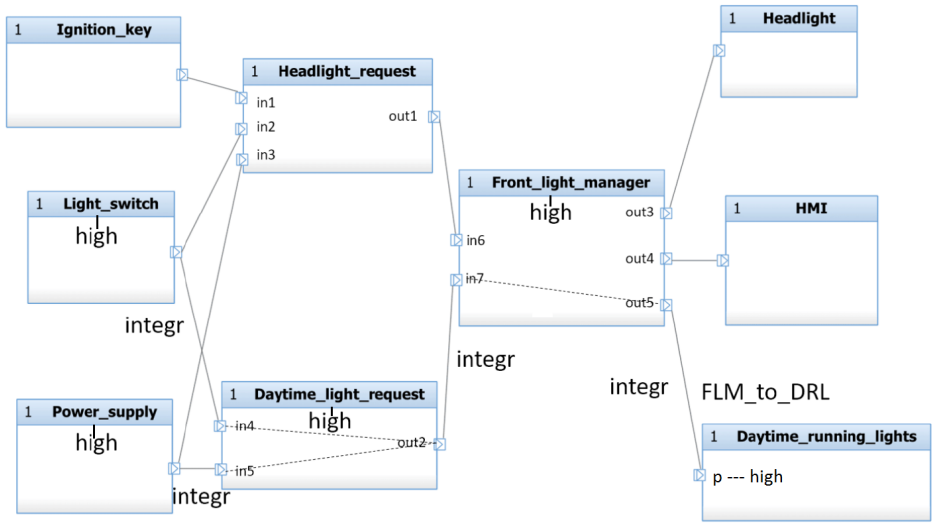

Fig. 14 Annotated model of the Front Light Manager.

the model and the data causal dependencies. As to accuracy, the data dependencies discovered with our approach are set at a level between the coarse grain approach of AUTOSAR and an exhaustive search of all possible executions.

The approach has been applied to the AUTOSAR Front Light Manager use case, using a prototype tool that implements the abstract execution of the runnables. In particular the application of the tool shows that the proposed approach provides annotated models that verify secure data flow property and reduce the number of security services, such as the number of encryption (or hash) operations invoked by components. The modular analysis of runnables makes the approach scalable with respect to the size of the system.

Further studies on automotive systems can be developed to improve the efficiency of the security services, for example it may be interesting to apply some modifications to the topology of the system to limit the number of paths from sensors to actuators connected to critical functions of the cars.

Acknowledgements The authors would like to thank the anonymous referees for their useful comments and suggestions.

References

1. A. Adelsbach, U. Huber, and A. Sadeghi. Secure software delivery and installation in embedded systems. In *Embedded Security in Cars*, pages 27–49. Springer, 2006.
2. AUTOSAR. https://www.autosar.org/.
3. AUTOSAR. General requirements on basic software modules. https://www.autosar.org/fileadmin/user_upload/standards/classic/3-2/AUTOSAR_SRS_General.pdf.
4. AUTOSAR. Safety use case example. https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_EXP_SafetyUseCase.pdf.
5. AUTOSAR. Specification of crypto abstraction library. https://www.autosar.org/fileadmin/user_upload/standards/classic/4-1/AUTOSAR_SWS_CryptoAbstractionLibrary.pdf.
6. AUTOSAR. Specification of crypto service manager. https://www.autosar.org/fileadmin/user_upload/standards/classic/4-1/AUTOSAR_SWS_CryptoServiceManager.pdf.
7. AUTOSAR. Specification of module secure onboard communication:. https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_SecureOnboardCommunication.pdf.
8. M. Avvenuti, C. Bernardeschi, N. De Francesco, and P. Masci. Jcsi: A tool for checking secure information flow in java card applications. *Journal of Systems and Software*, 85(11):24792493, 2012.
9. A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a java-like language. In *Proceedings of the 15th IEEE Workshop on Computer Security Foundations*, CSFW '02, pages 253–, Washington, DC, USA, 2002.
10. R. Barbuti, C. Bernardeschi, and N. De Francesco. Abstract interpretation of operational semantics for secure information flow. *Inf. Process. Lett.*, 83(2):101–108, 2002.
11. D.E. Bell and L.J. LaPadula. Secure computer systems: Mathematical foundation. In *MITRE Technical Report 2547, Volume I*, 1996.
12. C. Bernardeschi, N. De Francesco, G. Lettieri, and L. Martini. Checking secure information flow in java bytecode by code transformation and standard bytecode verification. *Software - Practice and Experience*, 34(13):1225–1255, 2004.
13. C. Bernardeschi, G. Del Vigna, M. Di Natale, G. Dini, and D. Varano. Using autosar high-level specifications for the synthesis of security components in automotive systems. In *Intl. Work. on Modelling and Simulation for Autonomous Systems*, pages 101–117. Springer, 2016.

14. C. Bernardeschi, M. Di Natale, G. Dini, and D. Varano. Modeling and generation of secure component communications in autosar. In *The 32nd ACM SIGAPP Symposium On Applied Computing*. ACM, 2017.
15. S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno, et al. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security Symposium*. San Francisco, 2011.
16. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 4(2):511–547, 1992.
17. P. J. Denning.. D. E. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 7(20):504–513, 1977.
18. IBM. Rational rhapsody. https://www.ibm.com/us-en/marketplace/rational-rhapsody/details.
19. J. Jürjens. UMLsec: Extending UML for secure systems development. In *UML 2002—The Unified Modeling Language*, pages 412–425. Springer, 2002.
20. S. Kehr, M. Pani, E. Quiones, B. Bddeker, J. B. Sandoval, J. Abella, F. J. Cazorla, and G. Schfer. Supertask: Maximizing runnable-level parallelism in autosar applications. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 25–30, 2016.
21. J. Kienberger, P. Minnerup, and B. Kuntz, S.and Bauer. Analysis and validation of autosar models. In *Proceedings of the 2Nd International Conference on Model-Driven Engineering and Software Development*, MODELSWARD 2014, pages 274–281, Portugal, 2014.
22. K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, et al. Experimental security analysis of a modern automobile. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 447–462. IEEE, 2010.
23. K.R.M. Leino and R. Joshi. A semantic approach to secure information flow. In *Proc. 4th International Conference, Mathematics of Program Construction, LNCS 1422*, pages 254–271. Springer Verlag, 1998.
24. K. Lemke, C. Paar, and M. Wolf. *Embedded security in cars*. Springer, 2006.
25. C. Lin and A. Sangiovanni-Vincentelli. Cyber-security for the controller area network (can) communication protocol. In *2012 International Conference on Cyber Security*, pages 1–7. IEEE, 2012.
26. T. Lodderstedt, D. Basin, and J. Doser. SecureUML: A UML-based modeling language for model-driven security. In *UML 2002–The Unified Modeling Language 2002*, pages 426–441. Springer, 2002.
27. G. Macher, M. Stolz, E. Armengaud, and C. Kreiner. Filling the gap between automotive systems, safety, and software engineering. *e & i Elektrotechnik und Informationstechnik*, 132(3):142–148, 2015.
28. MISRA. Guidelines for the Use of the C Language in Vehicle Based Software. Motor Industry Research Association, Nuneaton, 1998.
29. D. Mizuno, M. Schmidt. A security flow control algorithm and its denotational semantics correctness proof. *Formal Aspects of Computing*, 4(1):727–754, 1992.
30. M. Pani, S. Kehr, E. Quiones, B. Boddecker, J. Abella, and F. J. Cazorla. Runpar: An allocation algorithm for automotive applications exploiting runnable parallelism in multicores. In *2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, 2014.
31. C. Saad and B. Bauer. *Data-Flow Based Model Analysis and Its Applications*, pages 707–723. Berlin, Heidelberg, 2013.
32. M. Saadatmand and T. Leveque. Modeling security aspects in distributed real-time component-based embedded systems. In *Information Technology: New Generations (ITNG), 2012 Ninth International Conference on*, pages 437–444. IEEE, 2012.
33. A. Sabelfeld and A.C Mayers. Language-based information-flow security. *IEEE journal on selected areas in communications*, 21(1), 2003.
34. A. Sabelfeld and D. Sands. *A Per Model of Secure Information Flow in Sequential Programs*, pages 40–58. Berlin, Heidelberg, 1999.
35. W. Stephan, S. Richter, and M. Muller. Aspects of secure vehicle software flashing. In *Embedded Security in Cars*, pages 17–26. Springer, 2006.
36. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1992.
37. A. M. Wyglinski, X. Huang, T. Padir, L. Lai, T. R. Eisenbarth, and K. Venkatasubramanian. Security of autonomous systems employing embedded computing and sensors. *Micro IEEE*, 33(1):80–86, 2013.
38. S. Zdancewic and A. C. Myers. Secure information flow and cps. In *Proceedings of the 10th European Symposium on Programming Languages and Systems*, ESOP '01, pages 46–61, London, UK, UK, 2001.

APPENDIX A

This section reports the abstract rules for the instructions. In the rules we use the following notations:

- i is the `bb` of the CFG to which the instruction belongs.
- *lvar* is used for local variables, *gvar* for global variables, P for sender-receiver ports, *arg*& denotes arguments passed by reference, *ptr* is used for pointers and *array* for arrays.
- $f()$ is used for functions, including runnables.
- $Scope(\texttt{bb}_i)$ is a function that returns the set of blocks in the scope of the conditional instruction in $\texttt{bb}_i$.
- $A[\delta/x]$ is a global context equal to A except for the variable x that is assigned δ. Similarly, for other elements in the global context.
- $Q(M[\delta/x])$ is a local context equal to Q except for the variable x in memory M that is assigned δ.
- $Q(Env[\delta/Env])$ is a local context equal to Q except for the environment that is assigned δ.

Some rules regarding global variables are omitted, because they can easily be derived from the corresponding rule of local variable using the global context in place of the local memory. We note that, the level of variables and function's parameters, return and environment, in the global context file A never decreases. For example, if x is in the global context, the assignment of an expression to x updates the level of x to the *lub* between the current level and the level of the expression. If x is in the local memory the assignment

$$\textbf{Expr}_{const} \quad \frac{k \in const \quad Q_i = (M, \sigma)}{\langle k, \langle A, Q\rangle\rangle \longrightarrow_{expr} \sigma}$$

$$\textbf{Expr}_{var \in lvar} \quad \frac{x \in lvar \quad Q_i = (M, \sigma)}{\langle x, \langle A, Q\rangle\rangle \longrightarrow_{expr} M(x) \cup \sigma}$$

$$\textbf{Expr}_{var \in \{gvar \cup P\}} \quad \frac{x \in gvar \quad Q_i = (M, \sigma)}{\langle x, \langle A, Q\rangle\rangle \longrightarrow_{expr} A(x) \cup \sigma}$$

$$\textbf{Expr}_{*ptr \in lvar} \quad \frac{ptr \in lvar \quad Q_i = (M, \sigma)}{\langle *ptr, \langle A, Q\rangle\rangle \longrightarrow_{expr} M(ptr) \cup \sigma}$$

$$\textbf{Expr}_{array \in lvar} \quad \frac{array \in lvar \quad Q_i = (M, \sigma)}{\langle array[j], \langle A, Q\rangle\rangle \longrightarrow_{expr} M(array) \cup \sigma}$$

$$\textbf{Expr}_{op} \quad \frac{\langle e_1, \langle A, Q\rangle\rangle \longrightarrow_{expr} \delta_1 \quad \langle e_2, \langle A, Q\rangle\rangle \longrightarrow_{expr} \delta_2}{\langle (e_1 \ op \ e_2), \langle A, Q\rangle\rangle \longrightarrow_{expr} \delta_1 \cup \delta_2}$$

$$\textbf{Ass}_{var \in lvar} \quad \frac{x \in lvar \quad \langle e, \langle A, Q\rangle\rangle \longrightarrow_{expr} \delta}{\langle x{:=}e, \langle A, Q\rangle\rangle \longrightarrow \langle A, Q(M[\delta/x])\rangle}$$

$$\textbf{Ass}_{var \in gvar} \quad \frac{x \in gvar \quad \langle e, \langle A, Q\rangle\rangle \longrightarrow_{expr} \delta}{\langle x{:=}e, \langle A, Q\rangle\rangle \longrightarrow \langle A[A(x) \cup \delta/x], Q\rangle}$$

$$\textbf{Ass}_{var \in P} \quad \frac{x \in P \quad (x, y) \in L \quad \langle e, \langle A, Q\rangle\rangle \longrightarrow_{expr} \delta}{\langle x{:=}e, \langle A, Q\rangle\rangle \longrightarrow \langle A[A(x) \cup \delta/x; A(y) \cup \delta/y], Q\rangle}$$

$$\textbf{Ass}_{var \in arg\&} \quad \frac{x \in arg\& \quad \langle e, \langle A, Q\rangle\rangle \longrightarrow_{expr} \delta}{\langle x{:=}e, \langle A, Q\rangle\rangle \longrightarrow \langle A[f(\cdots, x \cup \delta, \cdots)b \cup \delta; \sigma / f(\cdots, x, \cdots)b; \sigma], Q\rangle}$$

$$\textbf{Ass}_{*ptr} \quad \frac{ptr \in lvar \quad \langle e, \langle A, Q\rangle\rangle \longrightarrow_{expr} \delta}{\langle *ptr{:=}e, \langle A, Q\rangle\rangle \longrightarrow \langle A, Q[M(ptr) \cup \delta/ptr]\rangle}$$

$$\textbf{Ass}_{array} \quad \frac{array \in lvar \quad \langle e, \langle A, Q\rangle\rangle \longrightarrow_{expr} \delta}{\langle array[j]{:=}e, \langle A, Q\rangle\rangle \longrightarrow \langle A, Q[M(array) \cup \delta/array]\rangle}$$

$$\textbf{If} \quad \frac{Q_i = (M, \sigma) \quad Scope = \{j_1, \cdots, j_n\} \quad \langle e, \langle A, Q\rangle\rangle \longrightarrow \delta}{\langle \texttt{if } e \texttt{ then goto } b_1 \texttt{ else goto } b_2, \langle A, Q\rangle\rangle \longrightarrow \langle A, Q_{j, j \in Scope}(Env[Env \cup \delta/Env])\rangle}$$

$$\textbf{Return} \quad \frac{\langle e, \langle A, Q\rangle\rangle \longrightarrow_{expr} \delta}{\langle \texttt{return } e, \langle A, Q\rangle\rangle \longrightarrow \langle A[f(a_1, \cdots, a_n)b \cup \delta; d / f(a_1, \cdots, a_n)b; d], Q\rangle}$$

$$\textbf{Goto} \quad \frac{}{\langle \texttt{goto } b_j, \langle A, Q\rangle\rangle \longrightarrow \langle A, Q\rangle}$$

$$\textbf{Invoke 1} \quad \frac{\langle x_j, \langle A, Q\rangle\rangle \longrightarrow_{expr} \delta_j \quad Q_i = (M, \sigma)}{\langle \texttt{f}(\texttt{x}_1, \cdots, \texttt{x}_n), \langle A, Q\rangle\rangle \longrightarrow \langle A[f(a_1 \cup \delta_1, \cdots, a_n \cup \delta_n)b; d \cup \sigma / f(a_1, \cdots, a_n)b; d], Q\rangle}$$

$$\textbf{Invoke 2} \quad \frac{x_j \in arg\& \quad Q_i = (M, \sigma) \quad f(a_1, \cdots, a_n)b; d \in A}{\langle \texttt{f}(\texttt{x}_1, \cdots, \texttt{x}_n), \langle A, Q\rangle\rangle \longrightarrow \langle A, Q[M(x_j) \cup a_j / x_j]\rangle}$$

$$\textbf{Invoke 3} \quad \frac{f(a_1, \cdots, a_n)b; d \in A}{\langle f(x_1, \cdots, x_n), \langle A, Q\rangle\rangle \longrightarrow_{expr} b}$$

of an expression to x sets the level of x to level of the expression.

When the abstract interpreter finds a function call it applies the three `invoke` rules in sequence:

- the first updates the global context with the levels of the actual parameters
- the second updates the variables passed by reference with the level in the global context
- the third evaluates the expression of the return of the function using the level in the global context

When the abstract interpreter finds an assignment to a sender/receiver port, which corresponds to a send operation, the abstract rule updates both the value of the sender port and the value of the receiver port in the global context A, using the set of links L.

When the abstract interpreter finds an assignment to a client/server port, this is transformed into a call to the runnable implementing the service. This rule is similar to a function call and it not shown in the figure.