Probabilistic Modeling and Simulation of Vehicular Cyber Attacks: An Application of the Meta Attack Language

Sotirios Katsikeas, Pontus Johnson, Simon Hacks, and Robert Lagerström

1KTH Royal Institute of Technology, Stockholm, Sweden
{sotkat, pontusj, robertl}@kth.se
2RWTH Aachen University, Aachen, Germany
hacks@swc.rwth-aachen.de

Abstract. Attack simulations are a feasible mean to assess the cyber security of systems. The simulations trace the steps taken by an attacker to compromise sensitive system assets. Moreover, they allow to estimate the time conducted by the intruder from the initial step to the compromise of assets of interest. One commonly accepted approach for such simulations are attack graphs, which model the attack steps and their dependencies in a formal way.

To reduce the effort of creating new attack graphs for each system of a given type, domain-specific attack languages may be employed. They codify common attack logics of the considered domain. Consequently, they ease the reuse of models and, thus, facilitate the modeling of a specific system in the domain. Previously, MAL (the Meta Attack Language) was proposed, which serves as a framework to develop domain specific attack languages.

In this article, we present vehicleLang, which can be used to design vehicles with respect to their IT infrastructure and to analyze their weaknesses. To model domain specifics in our language we rely on existing literature and verify the language using an interview with a domain expert from the automotive industry. To evaluate our results, we perform a Systematic Literature Review (SLR) to identify possible attacks against vehicles. These attacks serve as a blueprint for test cases checked against the vehicleLang specification.

Keywords: Domain Specific Language, Cyber Security, Threat Modeling, Attack Graphs, Vehicular Security

1 Introduction

Vehicles, especially passenger cars, are ubiquitous in European countries and their amount is still rising. For example, from January 2009 to January 2018 the number of registered passenger cars was increased by nearly 18% in Sweden [61] and 12.5% in Germany [33]. This means that 62% of Swedes and 56% of Germans owned a car in average in January 2018. Simultaneously, IT-systems
pervade vehicles more and more to increase driving security and user experience. Contemporaneously, the number of IT related security issues grows as well. This raises the need to assess the cyber security of vehicles.

However, assessing the cyber security of computing systems in general and of vehicles in special is difficult. In order to identify vulnerabilities, the security-relevant parts of the system must be understood, and all potential attacks have to be identified. There are three challenges related to these needs: Firstly, it is challenging to identify all relevant security properties of a system. Secondly, it might be difficult to collect this information. Lastly, the collected information needs to be processed to uncover all weaknesses that can be exploited by an attacker.

Hitherto, we have proposed the use of attack simulations based on system architecture models to support these challenging tasks. Our approaches facilitate a model of the system and simulate cyber attacks in order to identify the greatest weaknesses. This can be imagined as the execution of a great number of parallel virtual penetration tests. Such an attack simulation tool enables the security assessor to focus on the collection of the information about the system required for the simulations, since the first and third challenges are tackled by the simulation.

As the previous approaches rely on a static implementation, we propose MAL (the Meta Attack Language). This domain-specific language defines which information about a system is required and specifies the generic attack logic. As MAL is a meta language, no certain domain of interest, like vehicle security, is represented. Therefore, this work aims to create and evaluate a domain-specific, probabilistic modeling language for simulation of cyber attacks on modern connected vehicles, so called vehicleLang, (cf. Figure 1).

![Meta modeling hierarchy](image)

**Fig. 1.** Meta modeling hierarchy

To create vehicleLang, we follow the means of DSR (Design Science Research) as presented by Peffers et al. Therefore, we conduct a domain survey to extract a feature matrix containing all the security assets, the possible attacks and the corresponding defenses. This feature matrix serves as input to compile a detailed list of all the possible attacks following some widely used attack clas-
sifications and taxonomy frameworks. Subsequent, vehicleLang is constructed and evaluated. The created artifact is limited to the internal networks of the vehicles (i.e., CAN bus, FlexRay, and LIN bus) and the attacks that are related. However, it is scheduled to extend vehicleLang to other parts as well.

The remainder of the paper is structured as follows. Section 2 considers related work, followed by Section 3 describing the applied methodology. In Section 4 we present the MAL and its syntax to provide the framework for vehicleLang, which is presented with the core language in Section 5. The language itself is evaluated by a number of test cases depicted in Section 6. Finally, the paper is concluded in Section 7.

2 Related work

Our work relates to three domains of previous work: model-driven security engineering, attack/defense graphs, and car security. First, in model-driven security engineering were domain-specific languages for security analysis of software and system models defined as vehicleLang is one. Second, attack/defense graphs are applied as formalism for its analysis. Last, besides the fact that vehicleLang is in the domain of car security, the results of existing car security research are utilized for evaluation.

Model-driven security engineering induced a large number of domain-specific languages like UMLsec [26,27], SecureUML [34], SECTET [115], or STS-ml [60]. These languages facilitate the capability to model a system’s design according to components and their interaction. Furthermore, they enable to model also security properties such as constraints, requirements, or threats. They are build upon different formalisms and tools like the Unified Modelling Language and the Object Constraint Language. Model checking and searches for constraint violations are applied to conduct security analysis in these languages.

Apart from the languages mentioned before, there exist some security languages which do not support automated analysis purposes, such as CORAS [35], secureTROPOS [41], and SecDSVL [2]. They offer only the capability to model security relevant properties. An analysis needs to be conducted manually without any further support.

The concept of attack trees is commonly based on the work of Bruce Schneier [50,57]. They were formalized by Mauw & Oostdijk [37] and extended to include defenses by Kordy et al. [29]. As summarized in [30], there are several approaches elaborating on attack graphs, e.g. [22,24,49,68]. Elaborating on the theoretical achievements of the beforehand presented papers, different tools using attack graphs were developed. These tools mostly build up on collecting information about existing system or infrastructure and automatically create attack graphs based on this information. For example, MulVal [21,48] derives logical attack graphs by associating the vulnerabilities extracted from scans with a probability. This probability expresses how likely an attacker is to exploit the vulnerability successfully. MulVAL is extended by k-Zero Day Safety [66] to compute zero-day attack graphs. NAVIGATOR [7] considers identified vulnerabilities as directly
exploitable by the attacker, given that she has access to the vulnerable system. The TVA tool \cite{47} models security conditions in networks and uses a database of exploits as transitions between these security conditions. Similarly, NetSecuritas \cite{14} composes scanners output and known exploits to generate attack graphs and corresponding security recommendations.

A sub domain of attack graph modelling are probabilistic attack graphs, e.g., facilitating Bayesian networks. In \cite{13}, the authors apply the TVA-tool to generate attack graphs, transform them to dynamic Bayesian networks, and enrich them with probabilities using CVSS (Common Vulnerability Scoring System) scores. CVSS is also utilized by \cite{71} to model uncertainties in the attack structure, attacker’s actions and alerts triggering. In \cite{53} a genetic algorithm is used to produce a security mitigation plan, which takes Bayesian attack graphs as an input to estimate the security risk on network systems.

All before mentioned approaches have in common not to be model based. Therefore, we have combined attack graphs and system models in our previous work: CySeMoL \cite{59}, P2CySeMoL \cite{19}, pwnPr3d \cite{24} and securiCAD \cite{11}. The central idea of these works is to automatically generate probabilistic attack graphs from a given system specification. The attack graph serves as inference engine that produces predictive security analysis results from the system model.

To the best of our knowledge, there exists no research in the direction of probabilistic attack graph modelling within the domain of vehicle security. Therefore, we relate following to work elaborating on vehicle hacking. A first impression on the domain of vehicle hacking can be received by the work of Wolf \cite{69} and Smith \cite{58}. First, the authors give insights into vehicles’ IT architectures which contributes to the general construction of our instance of MAL. Second, they depict different ways to compromise various components of the vehicle under investigation.

More detailed attack scenarios are given by different authors e.g. Wolf et al. \cite{70}, Takahashi et al. \cite{64}, Cho and Shin \cite{6}. The first two work concentrate on attacking the bus of the vehicle. In \cite{70} the authors attack different bus and show their weaknesses as well as the weaknesses of future buses. To strengthen vehicle’s security they also propose a secured communication along the various buses. In contrast to Wolf et al., Takahashi et al. \cite{64} focus on LIN bus (Local Interconnect Network). They show possible attacks on this elementary vehicle communication system in different case studies. Furthermore, they contribute different countermeasures to rise vehicle’s security. The last work by Cho and Shin \cite{6} attacks ECU (Electronic Control Units) using DoS (Denial-of-Service). For this purpose, they facilitate the error-handling scheme of in-vehicle network to shut down or disconnect the ECU. Additionally, they suggest and evaluate approaches to detect and prevent their attack.

3 Methodology

DSR is a widely applied and accepted mean for developing artifacts in information systems (IS) research. It offers a systematic structure for developing
artifacts, such as constructs, models, methods, or instances. As our research objective indicates the development of an artifact, the application of a DSR is appropriate. We stick to the approach of Peffers et al. since it transpired as effective in former research. It is split up into six single steps and two possible feedback loops (cf. Figure 2).

![Design Science Research Process](image)

The first two steps of DSR are already presented within the introduction. To design our artifact we based our attack language on the MAL. To model domain specifics in our language we rely on existing literature as presented in related work and we verify it against a conducted interview with a domain expert from the automotive industry. Moreover, the modeled language is reviewed on a weekly to biweekly basis to ensure correctness and completeness.

To evaluate our artifact we applied test cases, which can be differentiated into two classes. First, there are unit tests to ensure that there are no mistakes in the implementation of specific attack steps and classes. Those are implemented by the researcher developing the language itself and a researcher developing a language related to vehicleLang. Second, we created integration tests based on an attack list, which was created applying a SLR (Systematic Literature Review) following the approach of Webster and Watson. Moreover, the test cases are an instantiation of the vehicleLang and, thus, serve as demonstration.

4 The Meta Attack Language

As already presented, this attack language relies on the MAL (Meta Attack Language). Therefore, this section presents the core parts of the grammar
(syntax) for MAL. For the formalism behind the language and details we refer to the original work.

MAL mainly consists of classes (e.g., \texttt{Car}), their instance (i.e., \texttt{myCar}), attack steps on classes (e.g., \texttt{Car.hijack}), and defenses on classes (e.g., \texttt{Car.immobilizer}). Furthermore, the entities of MAL are related to each other. Classes, and respectively their instances, can be linked to each other (e.g., \texttt{Garage.parked} = \{\texttt{Car}\}). Attack steps are connected to each other, thus the successful compromise of one step leads to the second step (e.g., \(e = (\texttt{Garage.open, Garage.parked.accessible})\)). Additionally, attack steps can be either of the type OR or AND, signifying either one of its parental steps is needed to elaborate on this step (OR) or all steps are needed (AND). Lastly, defenses are parenting attack steps which they hinder to be performed if they are TRUE (e.g., \((\texttt{Car.immobilizer, Car.drive}) \in E\))

To each attack step a probability distribution can be associated describing the expected time to perform this step, so called local time to compromise. Simulating different attackers behaviour on the modelled MAL attack graph allows to calculate the global time to compromise. This value provides a measure of how secure various points of the system are in respect to attack resilience. Further, it facilitates a quantitative way of comparing systems designs.

Classes containing attack steps constitute the core entities of a MAL specification. A class is specified as such:

```plaintext
class Garage {
    | open
    --> parkedVehicle.accessible
}
```

\texttt{Garage} is the name of the class, and \texttt{open} is the name of its unique attack step. \texttt{\|} symbolizes that the attack step is of the type OR. AND is represented by the symbol \&, while defenses are denoted by a \#. The arrow, \(\rightarrow\), signifies that the compromise of attack step \texttt{open} allows for an attack on the attack steps \texttt{parkingSpace.accessible}. \texttt{parkingSpace} is an association role, which is defined in a separate part of the MAL specification:

```plaintext
associations {
    Vehicle[parkedVehicle] 0-2 <-- Parking --> 1 [isParked] Garage
}
```

Association ends are bound to types with cardinalities like UML class diagrams. In this example, an object of the class \texttt{Garage} can be connected to up to two objects of the class \texttt{Vehicle}. Both ends of an association have roles, which are used for navigation. Thus, \texttt{myGarage.parkedVehicle} refers to the set of \texttt{Vehicle} objects connected to \texttt{myGarage}. Consequently, a compromise of the \texttt{Garage.open} attack step enables the attacker for an attack on the \texttt{accessible} step of the connected \texttt{Vehicle} objects, i.e. the \texttt{Garage's parkedVehicle}.

MAL features inheritance in a manner similar to other object-oriented languages. It allows to define abstract classes that are only intended for specialization, and never instantiated. Following, \texttt{Vehicle} is an abstract class, specialized
into the concrete classes Car and Bicycle, which inherit the attack steps and associations of Vehicle. Car adds a further required attack step key to be able to drive the Car.

```java
abstractClass Vehicle {
    | accessible
      -> drive
    | key
      -> drive
}

class Bicycle extends Vehicle {
}

class Car extends Vehicle {
    | key
      -> drive
    | accessible
      -> drive
}
```

Some steps may be accomplished without effort. For instance, as soon as a bicycle is accessible to the attacker, she is able to drive the bicycle. However, sometimes attack steps require a certain amount of time. Therefore, it is possible for each attack step to specify the required time. For example, the time to short-circuit a Car takes a mean of 12 minutes and a standard deviation of 5 minutes specified by a Gamma distribution. This can be specified as follows:

```java
class Car {
    & short-circuit [GammaDistribution(24, 0.5)]
}
```

Objects can be configured at the time of instantiation which is used for modelling defenses. For example, some instances of the class Car should have an immobilizer, while others do not have one. In this case, the defense Car.immobilizer may be introduced. # represent defenses and BOOLEAN values indicate defense’s status. Technically, each defense includes an attack step. If the defense is FALSE, then the associated attack step is marked as compromised at the time of instantiation. The following example facilitates effect’s comprehension:

```java
class Car {
    | accessible
      -> short-circuit
    & short-circuit
    # immobilizer
      -> short-circuit
}
```

If Car.immobilizer is false, then the attacker will be able to reach the AND attack step short-circuit as soon as she has reached accessible. If, instead, Car.immobilizer is true, then short-circuit will not be reached, as it’s compromise requires the compromise of both parents.
5 The Vehicular Cyber Attack Language

Beforehand, we presented the foundations underlying the MAL and its syntax. Following, we will present, first, a core language containing common IT entities, and, second, vehicleLang covering all the basic assets that comprise the internal vehicle’s network, from ECUs and vehicle’s networks up to applications and services running on top of them. The core (light blue) and the extensions made by vehicleLang (dark blue) are depicted in Figure 3. The elements in gray are also part of the core, but have been changed within vehicleLang.

5.1 Core Language

The core language consists mainly of entities representing Machine, Vulnerability, Account, Data, Dataflow, and Network. There are still more entities, but in the following, we will focus on the previously mentioned. A Machine represents attackable entities (i.e., hard- and software) of a system under investigation and, thus, are exposed by a Vulnerability. A Vulnerability can be exploited by a user represented by an Account. An Account can also be capable to read, write, and delete Data which is stored on a Machine. Two Machines can exchange Data facilitating a Dataflow on a Network.
Following we will explore those entities with more detail and depict how they and their interrelations are modelled. If one connects to a Machine, one can try to authenticate to get access to the Machine. Another way to get access to a Machine is to bypassAccessControl. If an attacker achieved access to a Machine, she can connect to all executees (i.e., executed Software on this Machine), requestAccess to all stored Data, and start denialOfService attacks, which can be executed against the executees or to denyAccess of certain Data. Without being authenticated the attacker is, first, able to compromise all privileges related to the connection and, second, to exploit the vulnerability related to an account to gain its privileges.

As far as the attacker reached the Vulnerability, she can try to exploit it. We assume that her time consumption to exploit the Vulnerability is given by an exponential distribution with a rate parameter of 10. A successful exploited Vulnerability leads to compromised privileges of all related Accounts.

An Account can be compromised by either exploiting a Vulnerability or authenticating (e.g., stealing the Credentials). A compromised account allows to overtake other Accounts (e.g., a super user can authenticate as each other account on a machine) and use their privileges. Moreover, an attacker can authenticate at other Machines, which use the same account. Last, an attacker is capable of reading, writing, and deleting Data on a Machine.

Data is a recursive entity as it can store other Data inside itself. Considering the basic attack steps of Data, there are three: read, write, and delete. To get to these attack steps, the attacker needs to requestAccess and the used Account must have granted the corresponding privileges (e.g., anyAccountRead). The attack steps read, and delete behave intuitively. In contrast, write allows also to delete Data and to tamper Data on the Dataflow. The attack step denyAccess is the result of a denialOfService and prevents the accessibility of the data.

A Dataflow is a logical communication between two Software applications. It allows several different attack steps. First, an attacker can eavesdrop on the Dataflow and, therefore, she can read the contained data. Second, an attacker can try to carry out a man-in-the-middle attack on the data flow. This leads to the control of the contained Data. In both cases, that Data may be encrypted and authenticated, thus preventing a breach of confidentiality and integrity. Besides reading, writing, and deleting, a manInTheMiddle allows also maliciousRequests and maliciousResponds.

A Network is a physical connection between Machines like Ethernet LANs, the Internet, and Wifi networks. Basically, it provides access to its Dataflows and enables the attacker to perform denialOfService, manInTheMiddle, and eavesdrop attacks.

5.2 vehicleLang

vehicleLang extends the core by adding Firmware, ECU, and VehicleNetwork. The Firmware, which is some kind of Software, runs on the ECU, which is derived from a Machine. Besides the “simple” ECU, there is also a GatewayECU which acts as gateway to the Network. Moreover, a GatewayECU offers the possibility
to activate a firewall and an IDPS to prevent certain attack steps. The ECUs are connected via the VehicleNetwork to exchange data with each other.

An ECU specifies any ECU, or controller in a vehicle. An ECU is an embedded system in a vehicle controlling one or more electrical system or subsystems like the anti-lock braking system. A vehicle can contain up to 80 ECUs [10]. We model it as an extension of the existing concept of a Machine, since an ECU offers many attacks that are not related to the abstract type of a Machine. If an attacker is connected to an ECU she can try to get access to the ECU, attemptChangeOperationMode, or to modify the Firmware. After compromising ECU’s Firmware, bypassingAccessControl, or properly authenticating, the intruder has access to the ECU. The access to an ECU can be utilized to changeOperationMode, gainLINAccessFromCAN [64], modify the Firmware, or execute a serviceMessageInjection on the related Services running on this ECU.

A changeOperationMode puts the ECU into diagnostics, which is usually used for service mechanics to seek for failures of the system, or into update mode, which is e.g. utilized to install a new Firmware. An ECU in these modes will no longer send messages and an attacker can imitate it if she is careful, thus, executing a bypassMessageConfliction. Furthermore, the changeOperationMode can be protected by the defense operationModeProtection. It either prevents diagnostics mode after vehicles starts moving or allows diagnostics mode only after some physical change is done on the vehicle (e.g. a physical switch pressed by the mechanic).

bypassMessageConfliction enables the attacker to inject forged service messages that could notify about vehicle’s fault or report fake status like speed or operation mode. This can lead to an unresponsive ECU (denialOfService) or allows to inject messages for the Services running on this ECU (serviceMessageInjection). It is also possible to defend against serviceMessageInjection by using message confliction mechanisms, which act like a host-based IDPS (Intrusion Detection and Prevention System) [28]. A host-based IDPS monitors and analyzes the internals of a computing system as well as the network packets on its network interfaces. Therefore, it observes the inbound and outbound packets from the device and alerts if suspicious activities are detected [41].

A GatewayECU is a specialization of an ECU that acts as a gateway on a vehicle. A GatewayECU contains additionally an inline IDPS and a firewall. An inline IDPS scans the passing through traffic of a network in real time and analyzes it following defined rules to decide if an attack is already running [41]. The firewall is modelled as a defense firewallProtection on the attack step bypassFirewall. Mainly, firewallProtection should be TRUE if there is a correctly configured firewall in place. Similar to this, the IDPS is modelled as defense on gatewayIDPS. If the defense is FALSE, the attacker can access the network unrestricted, thus, she can connect to all related VehicleNetworks.

Furthermore, a GatewayECU offers forwarding as the lightest interaction with the router, where the router simply re-transmits received messages. Vulnerabilities may, however, lead to compromise of the router as well as of the associated firewall. Therefore, forwarding leads to connect.
**Firmware** is a specialization of **Software** and is in our case related to the firmware running on an **ECU**. The attacker tries to upload **Forged Firmware**, since this leads to full access on the ECU and **messageInjection** capabilities to the connected **Networks**. The attacker has to successfully perform the **firmwareModification** so that the ECU will execute it [69]. During the **firmwareModification** the forged **Firmware** has to be validated. A firmware is usually equipped with a checksum based on the principle of digital signatures to empower the ECU to check if the **Firmware** is trustworthy. The attacker can either perform the **firmwareModification** by having the keys to sign it (**passFirmwareValidation**), by cracking the checksum (**crackFirmwareValidation**), or by exploiting the absence of verification (**bypassFirmwareValidation**). We put in a defense for **bypassFirmwareValidation** by code signing and verification during upload, the use of strong checksum functions and no distribution of the private keys for signing.

The **VehicleNetwork** extends the **Network** of the core language and serves as abstract layer containing the common attack steps and relations of its specializations **CANNetwork**, **FlexRayNetwork**, and **LINNetwork**. E.g., **accessNetworkLayer** implies the possibility to transmit messages over the network. But, it does not imply the possibility to listen to others’ traffic on the network, because the attacker is outside the **GatewayECU** but with a possibility to communicate in to the network. Therefore, she is able to **connect** to all related ECU, perform a **network-SpecificAttack**, or execute a **messageInjection**. However, the eavesdrop attack is still inherited from the **Network**.

Even if there is an IDPS in action, **maliciousRequests/maliciousResponds** can be applied directly, because **man-in-the-middle** attacks are not easily, or not at all, detected by an IDPS. Third, **injection** can be performed, but only from **VehicleNetwork** using a **messageInjection**. An **injection** means that an attacker can make **maliciousRequests** easily but must try hard to make **maliciousResponds**, because an IDPS may recognize those. Next, the attacker may tamper the **Data**, and, last, execute a **denialOfService**, which leads to the **deletion** of the transmitted **Data**.

A **maliciousRequest/maliciousRespond** allows to **connect** to the **Machine** on top of which the **Service** or **Client** is running, respectively. For the purpose of protection an IDPS can be deployed which enforces the attacker in some cases to bypass it. As this takes time, we use an exponential distribution with a rate parameter of 6.13 to estimate the time consumption [2006].

A **CANNetwork** represents the CAN bus network which is a soft real-time control network used for e.g. anti-lock breaking or engine control [69]. It is subject to two CAN related attacks: **exploitArbitration**, and **busOffAttack**. To **exploitArbitration** for message prioritization in **CANNetwork** can lead to invalidation of legitimate messages and allows tampering with the **Dataflow**. **exploitArbitration** is different from the **messageInjection** attack, because it allows direct malicious respond and request.

A **busOffAttack** exploits the error-handling scheme of in-vehicle networks to disconnect or **shutdown** not compromised **ECUs** [6]. The **busOffAttack** is easy to mount if the attacker has access to a **CANNetwork** or specializations of it,
because the attacker only needs to find messages which are sent periodically. Consequently, a costly reverse-engineering of messages or the network is not necessary. Additionally, an IDPS cannot differentiate between the injected messages and error messages. Therefore, an attacker who has access to a CANNetwork can apply directly a busOffAttack and a denialOfService to all connected ECU, no matter if there is an IDPS in place or not.

Cho and Shin [6] propose a protection against the busOffAttack, which is modeled with the defense busOffProtection. In this case, the busOffAttack cannot be performed against the ECU. In reality, this defense is two fold. First, more than 16 consecutive error frames indicate a busOffAttack. However, some errors can produce also so many consecutive error frames. Therefore, second, the ECU sends a message to the CANNetwork and observes if there will be another message with the same ID which is not allowed on a CANNetwork. Consequently, there must be a busOffAttack and the ECUs can be modified in a way that they do not got shut down.

The FlexRayNetwork is a hard real-time control network used for X-by-wire applications e.g. break-by-wire or emergency braking systems. It allows three special kinds of attacks: commonTimeBaseAttack, exploitBusGuardian, and sleepFrameAttack. The attacker sends for a commonTimeBaseAttack more than defined messages within one certain timeframe to make the whole network inoperable [69, p. 103]. Consequently, this leads to a denialOfService of the FlexRayNetwork.

The Bus Guardian is usually a part of a FlexRayNetwork connected ECU and regulates the access to the bus in sending messages to certain, defined time periods [62]. This Bus Guardian is utilized in the exploitBusGuardian attack for sending well-directed faked error messages to deactivate ECUs [69]. Therefore, on all FlexRayNetwork connected ECUs a shutdown can be performed. However, the Bus Guardian mechanism is hardened and, therefore, an attacker may need to spend much effort [43].

The sleepFrameAttack results also in the shutdown of all connected ECUs. For this purpose, the attacker sends well-directed forged sleep frames to deactivate power-saving-capable FlexRay ECUs [69]. This takes her an effort expressed by an exponential distribution with a key parameter of 10. If FlexRay power-saving is enabled, the sleepFrameAttack cannot be performed, which is modelled as defense powerSavingIncapableNodes.

A LINNetwork is a low-level non-real-time subnet which is used for e.g. door locking or light sensors. The attacker can easily gainLINAccessFromCAN from the ECU [31,54] on the LINNetwork and, consequently, perform the attack step accessNetworkLayer. Another possible attack on the LINNetwork is injectBogusSyncBytes. To perform this attack, the attacker sends frames with bogus synchronization bytes within a SYNCH field. This makes the local LIN network inoperative or causes at least serious malfunctions [69]. This leads in our model to a denialOfService of the LINNetwork.

A specific attack on the LINNetwork which exploits the error handling mechanism is the injectHeaderOrTimedResponse. If the attack is successful, it allows
to tamper the Dataflow \[64\]. To overcome the shortcoming of the LINNetwork Takahashi et al. \[64\] propose a protection against this attack, which is modelled as defense headerOrTimedResponseProtection in our model. Additionally, it is not easy to exploit this attack \[64\].

6 Evaluation

According to Hevner et al. \[18\], five methods are possible to evaluate the output of DSR: observations, analysis, experiments, tests, and descriptions. As developing vehicleLang is similar to developing source code, we opt for testing as evaluation method. We ground our decision on the fact that testing is widely spread in application development and commonly accepted as means to ensure that an application behaves as intended.

More concretely, we apply two different kinds of testing. First, we implement unit tests to ensure that vehicleLang behaves like we expect it to. To ensure our results, we apply, additionally, cross checking by another developer, who works also on a realization of MAL. This cross checking includes a revision of vehicleLang as well as the implementation of further unit tests to uncover unintended behavior. Second, we implement integration tests. Those integration tests rely on a compiled attack list created by SLR.

Our SLR follows the methodology of Webster and Watson \[67\]. The scope of the SLR is in line with the objective of this paper, in other words, identifying attacks on the internal networks of vehicles. Therefore, we searched for the terms “vehicle security”, “in-vehicle protocol security”, “CAN bus attacks”, “LIN bus attacks”, “FlexRay attacks”, “connected cars security”, and “automotive communication security” in title and abstract of articles in the Google Scholar\[2\], IEEE Xplore\[3\], and Springer Books\[4\] databases from 2006 to present. We scanned the abstracts of about 100 found results, filtered them with respect to their relevance to our research objective, and ended up with fourteen conference papers \[5\,6\,17\,23\,28\,34\,45\,46\,63\,65\,70\], three long papers \[9\,38\,40\], five scientific books \[8\,35\,39\,58\,69\], and one ENISA (European Network and Information Security Agency) report \[12\] relevant to our objective. We completed the SLR by conducting a back- and forward search which resulted in no more additional papers.

We scanned the identified literature and found 22 possible attacks on seven different classes. We relate to each class its possible attacks and give a short description of the attack steps. Furthermore, we refer to defenses which could be put in place to prevent the attack. We also kept track of the literature which mentioned the attack, and linked to the the test cases which ensure the related behavior.

\[2\] https://scholar.google.com
\[3\] http://ieeexplore.ieee.org/Xplore/home.jsp
\[4\] http://www.springer.com/gp/products/books
\[5\] The complete tables can be found in Appendices A.1 and A.2
The following provides a sample test case as depicted in Figure 4. It is based on a scenario where an attacker aims to perform a maliciousResponse on the OtherDataflow by using as entry point the physicalAccess to VehicleNetwork vNet1. It simulates a VehicleNetwork with two ECUs, one running a Service and one consuming it with a receiver acting as a Client. Each ECU is placed in its own Network and the Networks are connected to each other via a GatewayECU. On this GatewayECU the firewall and the IDPS are unfortunately disabled. The attacker initially has access to the Network of the ECU running the Service. Consequently, she can also compromise the Dataflows in this Network. However, she only reached (was able to connect to) the Services and the respective ECU, but did not compromise them. Same holds for the other Network, since the attacker can compromise the Network, because the firewall and IDPS are disabled, but she was not able to compromise the ECU and the corresponding Client.

![Fig. 4. Sample Attack on Vehicle’s Network](image)

To be more concrete, we give the implementation of the described test case in Appendix A.3 Listing 1.1. First, we create all entities contained in our test case (cf. lines 5 to 16). Second, we establish the connections between the entities like communication over a network or including each other (cf. lines 18 to 30). Third, we create an attacker (cf. line 32) and give her an entry point to the graph of entities (cf. line 33). Fourth, we start the simulation (cf. line 34). Last, we check with assertions if the attacker could or could not achieve different attack steps in our model.

7 Conclusions and Further Work

Assessing the cyber security of vehicles is becoming increasingly important as IT-systems and cyber-physical systems pervade vehicles more and more and,
simultaneously, the number of IT related security issues grows. Within this article, we developed the vehicleLang based on the MAL. vehicleLang will foster security analysts in the vehicle’s domain to model the vehicle and to focus on analyzing possible weaknesses. To model domain specifics in our language we rely on existing literature as presented in related work and we verified it against a conducted interview with a domain expert from the automotive industry. For the purpose of evaluation, we performed an SLR to identify possible attacks against vehicles. Those attacks served as a blueprint for test cases used to validate the vehicleLang specification.

Further work remains. First, we only included the internal networks of the vehicles (i.e., CAN bus, FlexRay, and LIN bus) and the connected components to them. Nevertheless, a vehicle comprises of further elements, which we did not model so far, like the infotainment system and the connectivity with the Internet.

Second, our studies have shown that the expected time consumption of attackers is mostly unresearched in the domain of vehicle cyber security. Consequently, future research should elaborate on uncovering these numbers to create more suitable models. As already mentioned, creating tests for feasible to evaluate an artifact, but, third, a real world evaluation of vehicleLang is still missing and should be conducted in further extensions of our work.

The last point is not related to vehicleLang itself, but to MAL. MAL relies on a static model. This allows a fast computation of possible attack paths. Unfortunately, the fast computation is bought by no dynamics in the model. Thus, a simulated attacker would not be able to create new entities during an intrusion. In other words, we are not able to consider e.g. newly created servers or additionally plugged entities. However, this sounds more problematic than it is, as most dynamic situations that are possible to model in a static model (e.g., by including the potential objects already from the start, but leaving them inactive until reached by the attacker).

References

12. enisa: Cyber security and resilience of smart cars: Good practices and recommendations. [Online; accessed 10-April-2018]


# Appendix

## A.1 Assets & Attacks

### Table 1: Covered Attacks

<table>
<thead>
<tr>
<th>Related Asset</th>
<th>ID</th>
<th>Attack Name</th>
<th>Short Description</th>
<th>Possible Defense/Obstacle</th>
<th>Test Cases</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECU</td>
<td>A1.1</td>
<td>Bypass Access Control</td>
<td>An attacker can bypass access control and authenticate to the machine through the diagnostics interface</td>
<td>None</td>
<td>TC 2</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>A1.2</td>
<td>Denial of Service (DoS)</td>
<td>When an attacker performs a DoS attack to the ECU. This leads to DoS on the services running and deny of access to stored data. This can even lead to unresponsive ECU</td>
<td>None</td>
<td>TC 1, TC 2</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>A1.3</td>
<td>Shutdown</td>
<td>When an attacker successfully switches off or takes offline a working ECU</td>
<td>None, but it requires significant effort</td>
<td>TC 9, TC 10</td>
<td>[39]</td>
</tr>
<tr>
<td></td>
<td>A1.4</td>
<td>Change operation mode</td>
<td>An attacker might put the ECU into diagnostics (if vehicle is moving slowly or is stopped) or even update mode (bootmode)</td>
<td>Prevent ECU from entering diagnostics mode after it started moving for first time. Allow diagnostics mode only after some physical change on car.</td>
<td>TC 17, TC 18, TC 19, TC 20</td>
<td>[39]</td>
</tr>
</tbody>
</table>
Table 1: Covered Attacks

<table>
<thead>
<tr>
<th>Related Asset</th>
<th>ID</th>
<th>Attack Name</th>
<th>Short Description</th>
<th>Possible Defense/Obstacle</th>
<th>Test Cases</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECU</td>
<td>A1.5</td>
<td>Bypass message confliction</td>
<td>An attacker can bypass message confliction protection mechanisms by changing ECU’s operation mode (i.e. no conflicts) and achieve message injection.</td>
<td>See A1.4</td>
<td>TC 20</td>
<td>[39]</td>
</tr>
<tr>
<td></td>
<td>A1.6</td>
<td>Message injection</td>
<td>An attacker can inject forged messages, to the services that are running on this ECU, that could for example notify about vehicle’s fault or report fake status (speed, operation mode, etc.). This can even lead to non-responsive ECU/DOS attack.</td>
<td>Communication protection (authorization and/or authentication), message confliction mechanism, IDPS.</td>
<td>TC 7, TC 25</td>
<td>[28]</td>
</tr>
<tr>
<td></td>
<td>A1.7</td>
<td>Firmware modification</td>
<td>An attacker could upload a custom firmware on the target ECU so that he can gain full access on the ECU and maybe on the connected network.</td>
<td>Firmware validation/signing mechanisms that prevent custom firmware uploads</td>
<td>TC 21, TC 22, TC 23, TC 28</td>
<td>[39]</td>
</tr>
<tr>
<td>Gateway ECU</td>
<td>A2.1</td>
<td>Bypass firewall</td>
<td>If the firewall is disabled, then the attacker can bypass it</td>
<td>Enable firewall</td>
<td>TC 31</td>
<td>-</td>
</tr>
<tr>
<td>Related Asset</td>
<td>ID</td>
<td>Attack Name</td>
<td>Short Description</td>
<td>Possible Defense/Obstacle</td>
<td>Test Cases</td>
<td>References</td>
</tr>
<tr>
<td>---------------</td>
<td>----</td>
<td>---------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------------------</td>
<td>------------</td>
<td>------------</td>
</tr>
<tr>
<td>Gateway ECU</td>
<td>A2.2</td>
<td>Bypass IDPS</td>
<td>If firewall is disabled, then the attacker can attempt to bypass the IDPS by carefully injecting messages to the network</td>
<td>Enable IDPS, and in general bypass requires some effort from the attacker</td>
<td>TC 31</td>
<td>[28,34]</td>
</tr>
<tr>
<td>A2.3</td>
<td>Denial of Service</td>
<td>An attacker can perform denial of service attack on the connected networks</td>
<td>None</td>
<td>TC 6</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Vehicle Network</td>
<td>A3.1</td>
<td>Eavesdrop</td>
<td>An attacker can eavesdrop the network and the transmitted dataflows</td>
<td>None</td>
<td>TC 8</td>
<td>[5]</td>
</tr>
<tr>
<td>A3.2</td>
<td>Man in the Middle (MitM)</td>
<td>An attacker can intercept and tamper with the network’s communications</td>
<td>None</td>
<td>TC 8</td>
<td>[52]</td>
<td></td>
</tr>
<tr>
<td>A3.3</td>
<td>Denial of Service (DoS)</td>
<td>An attacker can perform a DoS attack to the network</td>
<td>None</td>
<td>TC 9, TC 10, TC 11, TC 24, TC 31</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>A3.4</td>
<td>Message Injection</td>
<td>An attacker can inject forged messages to the network, that for example could notify about vehicle’s fault or report fake status (speed, operation mode, etc.)</td>
<td>Attacker must have physical access to the network layer</td>
<td>TC 25, TC 26, TC 27, TC 29, TC 32</td>
<td>[39,40,52,58]</td>
<td></td>
</tr>
<tr>
<td>Related Asset</td>
<td>ID</td>
<td>Attack Name</td>
<td>Short Description</td>
<td>Possible Defense/Obstacle</td>
<td>Test Cases</td>
<td>References</td>
</tr>
<tr>
<td>---------------</td>
<td>------</td>
<td>------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------Adamtexxclude here:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAN Network</td>
<td>A4.1</td>
<td>Exploit CAN's arbitration mechanism</td>
<td>By exploiting the arbitration mechanism for message prioritization in CAN bus, an attacker could achieve invalidation of legitimate messages and/or message tampering</td>
<td>Requires network access plus effort is needed from the attacker</td>
<td>TC 9</td>
<td>[35,39]</td>
</tr>
<tr>
<td></td>
<td>A4.2</td>
<td>Bus-off attack</td>
<td>An attacker can exploit the error-handling scheme to disconnect or shut down good/uncompromised ECUs</td>
<td>The defense proposed on literature, plus effort is needed from the attacker</td>
<td>TC 9</td>
<td>[6]</td>
</tr>
<tr>
<td>FlexRay Network</td>
<td>A5.1</td>
<td>Common time base attack</td>
<td>An attacker that emits more than needed SYNC messages within one communication cycle can make the whole network inoperable</td>
<td>None, but effort is needed from the attacker</td>
<td>TC 10</td>
<td>[69]</td>
</tr>
<tr>
<td></td>
<td>A5.2</td>
<td>Exploit FlexRay's Bus Guardian</td>
<td>An attacker can utilize Bus Guardian to send well-directed faked error messages to deactivate controllers. However, Bus Guardian is hardened so much effort is needed</td>
<td>None, but effort is needed from the attacker</td>
<td>TC 10</td>
<td>[43,69]</td>
</tr>
<tr>
<td>Related Asset</td>
<td>ID</td>
<td>Attack Name</td>
<td>Short Description</td>
<td>Possible Defense/Obstacle</td>
<td>Test Cases</td>
<td>References</td>
</tr>
<tr>
<td>---------------------</td>
<td>------</td>
<td>----------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>------------</td>
<td>------------</td>
</tr>
<tr>
<td>FlexRay Network</td>
<td>A5.3</td>
<td>Sleep frame attack</td>
<td>An attacker can send well-directed forged sleep frames to deactivate power-saving capable FlexRay controllers</td>
<td>Power-saving features can be disabled, plus effort is needed from the attacker</td>
<td>TC 10</td>
<td>[69]</td>
</tr>
<tr>
<td>LIN Network</td>
<td>A6.1</td>
<td>Inject bogus sync bytes</td>
<td>An attacker that sends frames with bogus synchronization bytes within the SYNCH field can make the local LIN network inoperative or cause at least serious malfunctions</td>
<td>None, but effort is needed from the attacker</td>
<td>TC 11</td>
<td>[69]</td>
</tr>
<tr>
<td>A6.2 Gain LIN access from CAN</td>
<td></td>
<td></td>
<td>An attacker can only gain access to the LIN subnetwork through a CAN-bus node</td>
<td>The CAN-bus ECU must first be compromised</td>
<td>TC 11</td>
<td>[31,54]</td>
</tr>
<tr>
<td>A6.3 Inject header or timed response</td>
<td></td>
<td></td>
<td>An attacker can exploit the error handling mechanism of LIN bus to inject forged headers or messages to the network, but in general it is not so easy</td>
<td>The defense proposed in literature, plus effort is needed from the attacker</td>
<td>TC 11</td>
<td>[64]</td>
</tr>
</tbody>
</table>
A.2 Test Cases

1. CoreMachineTest
   TC 1 testMachineAccess
   TC 2 testBypassMachineAccess
   TC 3 testSoftwareHostToGuest
   TC 4 testSoftwareGuestToHost
   TC 5 testMachineAccountDataRWD

2. CoreVehicleNetworkTest
   TC 6 testGatewayECUAccess
   TC 7 simpleServicekMessageInjection
   TC 8 testMitmNetwork
   TC 9 testCANNetworkSpecificAttacks
   TC 10 testFlexNetworkSpecificAttacks
   TC 11 testLINNetworkSpecificAttacks

3. CoreDataTest
   TC 12 testDataAccess
   TC 13 testDataflow1DataAccess
   TC 14 testDataflow2DataAccess

4. CoreVulnerabilityTest
   TC 15 testVulnerability
   TC 16 testSoftware

5. CoreEcuTest
   TC 17 testConnectEcuAttacks
   TC 18 testConnectEcuAttacks2
   TC 19 testAccessEcuAttacks
   TC 20 testAccessEcuAttacks2

6. CoreFirmwareTest
   TC 21 testFirmwareValidation
   TC 22 testFirmwareValidation2
   TC 23 testBypassFirmwareValidation

7. MessageInjectionTest
   TC 24 testNetworkMessageInjection
   TC 25 testServicekMessageInjection1
   TC 26 testServicekMessageInjection2
   TC 27 testNetworkMessageInjectionAfterVuln
   TC 28 testNetworkMessageInjectionAfterFirmwareUpload
   TC 29 testProtectedNetworkMessageInjection
   TC 30 testProtectedNetworkMessageInjection2

8. AdvancedNetworkTest
   TC 31 testDataflowWithFirewallAndIDPS
   TC 32 testGainLINaccess
A.3 Sample Attack

Listing 1.1. Source Code of Sample Attack

```java
public void testDataflowWithFirewallAndIDPS() {
    boolean firewallStatus = false;
    boolean idpsStatus = false;

    ECU SrvEcu = new ECU("ServiceECU", true, true);
    ECU ClnEcu = new ECU("ClientECU", true, true);
    GatewayECU GateEcu = new GatewayECU("GatewayECU",
                                         firewallStatus, true, true);
    VehicleNetwork vNet1 = new VehicleNetwork("vNet1");
    VehicleNetwork vNet2 = new VehicleNetwork("vNet2");
    Dataflow dataflow = new Dataflow("Dataflow");
    Dataflow otherDataflow =
        new Dataflow("OtherDataflow");
    NetworkClient client = new NetworkClient("Client");
    NetworkService service =
        new NetworkService("Service");

    SrvEcu.addVehiclenetworks(vNet1);
    ClnEcu.addVehiclenetworks(vNet2);
    GateEcu.addVehiclenetworks(vNet1);
    GateEcu.addVehiclenetworks(vNet2);
    SrvEcu.addExecutees(service);
    ClnEcu.addExecutees(client);
    vNet1.addTrafficGatewayECU(GateEcu);
    vNet2.addTrafficGatewayECU(GateEcu);
    vNet1.addDataflows(dataflow);
    vNet2.addDataflows(dataflow);
    vNet2.addDataflows(otherDataflow);
    client.addDataflows(dataflow);
    service.addDataflows(dataflow);

    Attacker attacker = new Attacker();
    attacker.addAttackPoint(vNet1.physicalAccess);
    attacker.attack();

    dataflow.maliciousRequest
        .assertCompromisedImmediately();
    GateEcu.forwarding
        .assertCompromisedInstantaneously();
    GateEcu.bypassFirewall
        .assertCompromisedInstantaneously();
    GateEcu.gatewayNoIDPS
```
assertCompromisedInstantaneously();
GateEcu.gatewayBypassIDPS.assertUncompromised();
otherDataflow.maliciousRespond
  .assertCompromisedInstantaneously();
SrvEcu.connect.assertCompromisedInstantaneously();