Peter Dencker et. al.(Hrsg.): Automotive Safatey & Security 2017,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 103

Adapting Organic Computing Architectures to an Automotive Environment to Increase Safety &Security

Kevin Lamshöft, Robert Altschaffel, Jana Dittmann [1]

Abstract: Modern cars are very complex systems operating in a diverse environment. Today they incorporate an internal network connecting an array of actuators and sensors to ECUs (Electronic Control Units) which implement basic functions and advanced driver assistance systems. Opening these networks to outside communication channels (like Car-to-X-communication) new possibilities but also new attack vectors arise. Recent work has shown that it is possible for an attacker to infiltrate the ECU network insides a vehicle using these external communication channels. Any attack on the security of a vehicle comes implies an impact on the safety of road traffic. This paper discusses the possibilities of using architectures suggested by Organic Computing to reduce these arising security risks and therefore improve safety. A proposed architecture is implemented in a demonstrator and evaluated using different attack scenarios.

Keywords: Automotive, System Architectures, Organic Computing

1 Introduction

Modern cars are complex systems containing a wide array of actuators, sensors and ECUs (Electronic Control Units). Some of these components are essential for the basic function of a car while others provide assistance or amnesties to the driver. All these components are connected to an internal network. With the growing number of external means to connect to this network, like Car-to-X-Communication (C2X), In-Car Internet or remote diagnostics an inherent risk of attacks on these networks arise. Recent work [MV15] has proven this assumption. Any attack on the security of a vehicle carries the same implication on the safety of road traffic as error and faults of individual vehicular components. They can lead to dangerous situations either through direct means (e.g. failure of brakes), interruption of an assistance function the driver relies on (e.g. ABS) or distraction (e.g. Multimedia). The complex interplay of all these components is bound to further increase with the introduction of autonomous vehicles. This complexity challenges classical engineering approaches. Hence this paper explores the possibilities of using

[1] Otto-von-Guericke Universität, AMSL, Universitätsplatz 2, 39102 Magdeburg,
Kevin.Lamshoeft|Robert.Altschaffel|Jana.Dittmann@iti.cs.uni-magdeburg.de

architectures derived from the field of Organic Computing in order to cope with the growing complexity.
During this paper section 2 handles a brief introduction on the specifics of automotive IT and gives an overview on the topic of Organic Computing in general and the observer/controller architecture in particular. Section 3 will discuss how the naive implementation of Organic Computing architectures would perform in an automotive environment while section 4 presents and discusses changes to such an architecture proposed by us. Section 5 describes the implementation of a demonstrator used for evaluation of the concepts proposed in this work. Section 6 discusses practical scenarios in which an attacker injects spoofed messages in a malicious manner. Here it is evaluated if and how the demonstrator could reduce the impact of an attack. Section 7 concludes this paper with summary and outlook.

2 State of the Art

This section gives a brief overview on the state of automotive IT focused on the main vehicular network - the CAN bus. Further introduction is given on the principles of Organic Computing (OC) and the means to achieve them by using various architectures. Finally, information on anomaly-based intrusion detection will be presented since it will be used in the demonstrator used for evaluating the concepts presented in this work.

2.1 Specifics of automotive IT

Modern cars consist of dumb and smart components. The dumb (or passive) components are all parts without electronics. Smart (or active) components consist of:

- **Sensors** measure the conditions of the vehicle's systems and environment (e.g. pressure, speed, rain intensity etc.) but can also capture user input requests.
- **Actuators** are units that perform a mechanic actuation.
- **Electronic Control Units (ECUs)** perform the electronic processing of input signals, which are acquired via different types of sensors and relay commands to the actuators. Some units control critical systems of the vehicle such as the engine and safety systems (ABS, Airbag) while others control comfort units such as the door

control units. The number of ECUs embedded with a vehicle is still rising to more than 100 in 2010 [Su14].

- **Direct analogue cable connections** are used to carry measured signals (from sensors to ECU) or actuation impulses (from ECU to actuators).
- **Shared Digital Bus Systems** are used for communication among ECUs. Beyond the direct connections between ECUs and sensors/actuators, ECUs are additionally connected amongst each other via digital field bus systems [Tr09]. This shared medium is used to exchange required information, like forwarding digitized sensor signals, exchanging current operating parameters, remote actuation requests or diagnostic requests for maintenance purposes. In modern cars, several different technologies for digital automotive field bus systems are used. The most common automotive field bus system is the Controller Area Network (CAN) [Bos91], which is the core network of the vehicle systems communication. This CAN network is divided into sub-networks such as powertrain/engine, diagnostics, comfort or infotainment. ECUs are connected to each sub-network depending on their functions and these sub-networks interconnect in a ECU device called CAN Gateway which handles the routing of messages to different sub-networks. The specific sub-networks offer a shared medium for the exchange of CAN messages. The CAN message consists of several flags without further importance to this paper, the CAN ID and the payload. The CAN ID represents the type of a message and implies a certain sender and receiver for the message - hence any ECU on the specific bus will receive all messages but discard the ones with CAN ID unimportant to it. It is assumed that a message with the corresponding ID is send by the ECU normally responsible for this message. In addition, the CAN ID serves as priority. Since there is no sender verification it is very easy to insert crafted packets into CAN networks once access to an entity able to send on the bus (an ECU or a tap) is established.

Recent trends show the increased communication between cars and other cars (Car-to-Car, C2C) or infrastructure (Car-to-Infrastructure, C2I [Ka08]). These communication channels not only increase functionality but also carry the risk of new attack vectors, as demonstrated [MV15].
In an automotive environment, an attack on the security of the car is bound to also become a safety risk due to the already dangerous nature of road traffic. If in classical desktop IT a system crashes the system simple crashed. If the IT system in a moving vehicle crashes you have a moving vehicle that does not act reliable to user input and might as well crash in a more dramatic manner.

2.2 General Introduction to Organic Computing

Organic Computing is an approach to deal with the complexity regarding systems of systems. With the ever-growing amount of different participating systems, scenarios and circumstances classical engineering approaches are reaching their limits. OC aims at viewing the system of systems as an organic whole with the user only formulating aims or tasks while the subsystems themselves deal with the realization of these tasks, as evident in the quote from [Wu08]:
In organic computing, the only task humans hold on to is the setting of goals. As the machine is autonomously organizing, detailed communication between programmer and machine is restricted to the fundamental algorithm, which is realizing system organization.

A fundamental aspect of OC is emergence. [WH05] define emergence as:
A system exhibits emergence when there are coherent emergents at the macro-level that dynamically arise from the interactions between the parts at the micro-level. Such emergents are novel w.r.t. the individual parts of the system.

Hence emergence describes a macro behaviour of a complex systems not inherent in the behaviour of the specific components. A system based on OC principles is henceforth called an organic system and should fulfil several properties. These properties are known as self-x-properties. These include self-adaption as the core property of any organic system [MSU11]. Other examples for self-x-properties in their work are include self-configuration, self-optimization and self-healing, which are specialized types of self-adaption. Furthermore, self-perception was identified as a basic requirement for organic systems [Al14].

2.3 Generic Observer/Controller Architecture in Organic Computing

In order to guarantee that the macro behaviour of a decentralized self-organizing system meets the intended purpose a so called observer/controller is used. A suggestion for this central concept is the Generic Observer/Controller Architecture as introduced in [Ri06] and is

used the foundation for the approach presented in this work. The architecture consists of three major entities: A multi-agent system, called System under Observation and Control (SuOC), an observer and a controller. The Observer is monitoring the SuOC with sensors, processes the data and passes accumulated information on the state to the controller which then evaluates possible actions and might control the SuOC. The observer/controller pattern is built on top of the SuOC - if the observer/controller fails, the SuOC will retain its self-organizing structure.

The observer consists of different modules, which define the observation process:

- **O1 Monitor**: gains raw data from the underlying SuOC
- **O2 Log file**: saves data from each iteration which might be used for predictions
- **O3 Pre-Processor**: prepares data for analyses and prediction
- **O4 Data analyser**: applies a set of detectors on the pre-processed data; result is reflecting the current state of the SuOC
- **O5 Predictor**: predicts future system states based on raw data, history data and analyser data
- **O6 Aggregator**: accumulates data which is then passed to the controller

The controller receives aggregated data from the observer and compares it to the goals for the organic system by the external user. This component directs emergent behaviour of the SuOC in order to achieve desired emergent behaviour or disrupt or prevent undesired emergent behaviour. Three types of control can be applied by the controller: Influencing local decision rules, influencing the system structure and influencing the environment. The controller uses an internal action selector which selects best suited action based on the current situation of the SuOC (mapping) and forwards this decision towards its actuators. The applied action is saved in a history file and is in a next step evaluated by comparing the new system state with the state before. Depending on how much the action influenced the system state of the SuOC the fitness value for the mapping is updated. Hence the mapping is improved over time. This learning process can be enhance using machine learning techniques, for example, by using evolutionary algorithms, learning classifier systems, reinforcement learning or neural networks.

2.4 Anomaly-based Intrusion Detection

The use of use Shannon entropy for detection of anomalies in in-vehicle networks was proposed in 2011 [MN11]. This work uses entropy which is the expected average value of information that a message carries in a message flow. Entropy can be calculated for a single message, specific CAN IDs or the whole bus traffic. The entropy of the usual behaviour is determined a priori in a learning phase. For intrusion detection, the entropy is calculated in fixed intervals and compared to the entropy of normal behaviour. An anomaly is found if the difference if the expected and the current value differ more than a threshold. This approach has been successfully tested and proved useful [MSGC16]. An Extension [CK16] proposes the use of fingerprinting techniques known from conventional IT in automotive networks. Here the clock skew estimation is used to localise affected ECUs.

3 Adopting the Generic Controller/Observer Architecture to Automotive Bus Networks

As mentioned in section 2.1 defending against advanced attacks on automotive bus networks is a non-trivial task.
However, most attacks have one aspect in common: On the lowest level, they are sending forged messages on automotive bus networks. Such a message does look legitimate in CAN bus networks since there is no authentication. While fuzzing and replay attacks reportedly can be detected in parts by automotive IDS [MSGC16], more sophisticated, targeted attacks can be recognized only by looking at the result in the overall system – in this case the car.

For example, prior experiments with several cars have shown that an attacker is able to lock the doors permanently by sending forged messages on the CAN bus. This prevents the passengers from leaving the car. In addition, the heating can be turned on and air conditioning can be turned off while preventing user input. This can lead to serious disturbance or even bodily harm. For this attack, an attacker only needs a few forged messages. Each message by itself is inconspicuous and seems legitimate and therefore

should not raise any alarm. However, the behaviour of the overall system, caused by these accumulated messages, is suspicious and harmful.

As shown before a car is a system of system and hence this undesired behaviour akin to an undesired emergent behaviour. As shown in section 2.3 the Generic Observer/Controller Architecture forms the part of the Organic Computing approach aiming to observe emergent effects and either taking actions to achieve desired emergence or preventing/disrupting undesired emergence. By adapting this architecture to the requirements of cars and implementing the concept we might be able to detect such advanced attacks and mitigate their impacts with low-cost hardware. In contrast to Intrusion Detection Systems the presented approach goes further and does not only detect anomalies but also takes actions to counter attacks. This is achieved by not only looking for anomalies but aggregating data from multiple sources in order to get a better understanding of the systems state and reducing the number of false-positives. By using methods derived from the field of machine learning the system learns with each incident and gets better over time.

The following section will deal with adapting this generic approach to the automotive domain.

3.1 Theoretical Considerations

The first step on adapting the Generic Observer/Controller Architecture to the automotive domain is a the definition of system boundaries for the SuOC. Since this approach aims to increase robustness against attacks, or in fact general malfunctions as well, we want to achieve desired and disrupt undesired emergent behaviour of the whole system – technically speaking the whole car. Considerations on the feasibility of defining a car as SuOC rely on the definition of a SuOC presented in section 2.3. Here a SuOC is defined as a multi-agent, self-organizing system.

Automotive IT consists of a network of ECUs, sensors and actuators. As these ECUs are autonomous entities, communicating and interacting with each other, observing and acting in an environment to achieve goals they can be considered agents. Technically speaking a set of ECU networks can

therefore considered a multi-agent system. Following the definition given by [MWJ+07] a self-organizing system is self-managing, structure-adaptive and employs decentralized control. The network of ECUs is self-managing in the sense of adapting to different I/O requirements without an explicit external control input on how to achieve this. The driver gives a general objective (I/O requirement) towards the car (e.g. Acceleration) causing multiple ECUs to work together in order to achieve that goal. Automotive IT is structure-adaptive as the ECUs maintain their structure and provide the systems primary functionality. As there is no central ECU that controls the others it employs decentralised control – leading to the conclusion that the network of ECUs can be seen as a self-organizing system.

3.2 Adapting the Observer to Automotive Bus Networks

We aim at not only detecting irregularities in the function but the car but also on reaction towards these irregularities. Therefore, we propose the usage of multiple cooperating Observer/Controller instances (as defined in the Generic Observer/Controller Architecture). One major task of the observer is, similar to IDSs known from conventional IT, detecting anomalies - called "symptoms" in the MAPE cycle [IBM06]. These symptoms might lead to an undesired emergent behaviour. Going beyond the possibilities of an IDS in this approach the Observer considers the symptoms more thoroughly by aggregating data from several data analysers (resp. emergence detectors or IDSs), Log files, Predictors and communicating with other Observers. This helps to reduce false-positives and get a better understanding of the symptom before applying control actions.

In order to achieve this the Observer presented in section 2.3 is adapted to an automotive environment as follows:

- **O1 Monitor**: In the OC architecture the observer is monitoring the influence of the agents on their surroundings. In the case of automotive IT this means that the Observer is not monitoring the ECUs directly but their influence on the car. Hence this means the monitoring of specific actuators. As it would require a broad range of sensors to monitor the actual behaviour of all actuators we propose a different approach in not monitoring the physical actions of the actuators but in monitoring the communications on the automotive bus networks. This is feasible since actuators

at this moment do not have any computing power themselves and just act on the orders given by them from the ECUs. Hence, if there is an action, there is also a message causing this action. This might not hold true for the future, though, so future work will add sensors which directly monitor the physical actions of the actuators. Even then the primary sensor will most likely still be the networking interface which allows the observer to read all BUS communication.

- **O2 Log file**: Since the log files function is basically to save data for further iterations and predictions no adaption to an automotive context is needed.
- **O3 Pre-Processor**: This step prepares the raw data supplied by the monitor for the following steps of analysing and prediction. Depending on the Data Analysers and Predictors the specific tasks of the Pre-Processor might vary. For bus networks the main part of the Pre-Processor is filtering (e.g. leave out keep-alive-messages or duplicates) and prioritising (e.g. error messages) of the network traffic. Aggregation and counting of reoccurring messages might be useful as well.
- **O4 Data Analyser**: This is one of most important parts in the approach presented in this work. It applies a set of detectors on the pre-processed data in order to identify undesired behaviour of the car. One trivial approach would be to identify error frames (e.g. failure of signal lights) on the bus network. This relies on the affected ECU detecting such failures. It hence would not work in scenarios where no malfunction of specific components are caused but rather a harmful macro behaviour, like in the scenario introduced in section 3. Therefore, a deeper analysis of the network traffic is needed in order to detect undesired emergent behaviour. Two different approaches are suggested here: detect unusual behaviour and detect illogical behaviour and detect rule violations. An example for unusual behaviour might be the toggling of certain features multiple times in short intervals (e.g. recurring signals to close windows). This might also point to a component failing to react on a given input, like repeated pushing of a brake pedal without reaction of the brake. In these cases further analysis will be conducted. As the log file stores information about the system state the Analyser is able to check if the speed reduced in case of the repeatedly triggered brake pedal. Examples for illogical behaviour are opening the trunk while driving, pushing brake and accelerator pedals at same time causing invalid system states. The third option is to define correct behaviour a priori by rules (e.g. doors need to be closed while driving) and monitor violations.
- **O5 Predictor**: Based on data from the analyser and log file the predictor calculates possible future system states which will be passed to the controller. This enables the controller to take preventative actions. For example, the predictor extrapolates the speed for a time t+1, based on the current speed at time t and the speed of the state before at time t-1.
- **O6 Aggregator**: The aggregator accumulates information of the analysers detectors and the predictor to give a most accurate evaluation on the current and future system states to the controller which then based on that information takes actions to influence the environment towards a desired behaviour. In order to identify and

analyse the symptom the Aggregator needs to interpret the data coming from the data analysers. Hence a semantic lookup table (which message is representing what information) is needed but could also be implemented at an earlier stage in the Data Analyser. The data coming from the Observer needs to be encoded in a way that the Controller can map it to an action. One naive approach is to pass raw information coming from the analysers without any semantics towards the Controller. For example, the corresponding CAN IDs which show anomalies could be mapped to an action (e.g. a detected anomaly at 0x172 would be mapped to an action which opens the windows). Depending on the Data Analysers and information gathered by the Aggregator more precise information could be passed. An exemplary data set for the given scenario could be *state = [{Entropy Anomaly Detector -> Affected ID: 0x172}, {ECU Fingerprint Anomaly Detector -> Anomaly Source: central lock ECU}, {Aggregator -> Result: doors do not open]* and would map *actions = [{flash central lock ECU}, {open windows}, {open trunk}]*. We recommend using multiple Observers communicating with each other, getting additional information and do cross checking, to specify the systems state more accurately. For example, an Observer monitoring unusual behaviour of the doors (e.g. a door is opening and closing repeatedly) can request additional information from another Observer which is monitoring different parts of the vehicle like the powertrain bus in order to determine if the vehicle is moving.

3.3 Adapting the Controller to Automotive Bus Networks

The Controller's main task is to take actions based on information it receives from the Observer. The Controller takes actions by sending messages to the agents (ECUs) which then applies actions to the car. Therefore, a mapping of the environmental states and actions is required by the Controller.

When the Controller gets information by the Observer it applies a set of classifiers (rules) to map environmental states into actions. Each classifier has a fitness value/reward that defines the quality of the action and is updated when the Controller gets feedback on how good the applied action has performed.
If multiple classifiers fit a given environmental state the one with highest fitness/reward is selected. The structure of a classifier is given by*{State -> Action : Fitness}*. One basic example for a classifier would be {*0x172 -> 0x172#1122 : 42*}, where *0x172* marks the CAN ID, which shows anomalies, *0x172#1122*the action (the CAN message to be sent by the controller) and *42* the fitness value. Before applying actions, we suggest

reporting to the driver. The user should be informed that an anomaly has been detected, what causes are probable and which influences are detected and expected. The driver should have the option to mark the anomaly as false-positive. Causes for false-positives are found in the data analysers as well as in unexpected behaviour of the driver or passengers. In a next step the driver should be informed on planned actions and asked for permission. There might be situations in which the driver has to act by himself (e.g. applying manual handbrake to slow the car in case of brake failure). In that case, the controller only gives a warning and recommendations on how to act. The rules for reaction can be added manually or generated by machine learning.

4 Implementation

The approach presented in this paper has been evaluated in a demonstration in order to examine its merits and drawbacks. For our experimental work we used the electronics of an Audi Q7 built in 2008. These electronics have been extracted after a crash test. Due to the crash, most parts of its drive train were missing. Hence, we focused on ECUs communicating on the comfort bus. The CAN bus was accessed directly by a Raspberry Pi using CANtact interfaces. CANtact was chosen for using the SocketCAN driver of can-utils which allows to receive all frames from the bus and send arbitrary messages. A Raspberry Pi was used for implementing the Observer/Controller functionality.
The following subsections give an overview on how the concepts developed in section 3 have been implemented for the demonstrator.

4.1 Observer Implementation

This subsection describes how the observer was implemented in the demonstrator.

- **O1 Monitor:** The Monitor was implemented on a Raspberry Pi using Python. It uses a SocketCAN interface to communicate with the CANtact board and the python-can package to monitor traffic on the bus.
- **O2 Log File:** The demonstrator saves general information on the vehicle state, e.g. the status of ignition and door locks.

- **O3 Pre-Processor:** Received CAN frames are striped to CAN ID and payload only, removing timestamps since these would cause bogus results during entropy calculation.
- **O4 Data Analysers:** An entropy-based anomaly detection approach was implemented. (see section 2.4). The demonstrator implements a combination of entropy calculation for the whole bus and dedicated calculations for specific IDs. The entropy analyser has to go through a learning phase before it can be used. In that learning phase the average entropy μ, standard deviation σ, a model parameter k and time windows t are calculated and defined. A target space which marks the usual behaviour as a range$[\mu - \sigma, \mu + \sigma]$and a acceptance space which allows minor deviation of the usual behaviour (reduces false-positives) as range $[\mu - k\sigma, \mu + k\sigma]$, where k is a model parameter are defined. Figure 1 shows the learned parameter settings used in the experiments.

Entropy Calculation	Controller Area Network ID	average entropy μ	standard deviation σ	model parameter k	time / message window t
Comfort Bus	-	5.94	0.1	4.7	5 (seconds)
Locks	0x171	1.385	0.252	1.9	30 (frames)
Windows	0x182	0.11	0.33	0.6	30 (frames)
Multimedia	0x5C4	0.941	0.02	1.95	30 (frames)

Fig. 1: Learned parameter settings for the demonstrator

- **O5 Predictor:** Prediction is not included in this demonstrator.
- **O6 Aggregator:** The Aggregator fetches data from the entropy analysers and general information of the log file and passes them towards the Controller module. In this demonstrator the Observer reports where an anomaly is detected (whole bus and/or CAN IDs), the value of constraint violation (distance of measured entropy from acceptance space) and a human readable state of the system (e.g. *car standing, doors locked, anomaly regarding locks detected*) in order to inform the driver and for debugging purposes.

4.2 Controller Implementation

For the implementation of the controller a pre-defined classifier set is used. Using fitness values and wildcards for the classifiers the controller is able to map any state given as input from the Observer. Before taking any action the controller reports the aggregated information of the observer and the mapped action to the user (in the current version via CLI) and asks for permission. After taking the action the user is asked whether the problem

persists (in future versions the controller evaluates that by itself using data from the observer). If the problem is solved the classifier gets a reward which increases its fitness. If the anomaly persists the controller takes another matching action if available. If that is not the case the controller notifies the user to stop the car.

5 Evaluation

This section describes how the demonstrator was evaluated.

5.1 Adversary Model and Attack Scenarios

Prior research has shown several physical and remote attack surfaces and vectors [MV15]. In this evaluation scenario we selected an attacker who is able to send arbitrary messages with spoofed IDs. This allows the attacker to toggle certain features of the car in with the aim of making driving impossible or at least very uncomforting. This type of attack is typical for ransomware campaigns from other computational domains. We define three different attack scenarios for our experiments:

- **S1 Lockout** The attacker locks the driver out of his car. This attack can be implemented for the Q7 with a minimum effort. When the car gets locked or unlocked by the remote a corresponding sequence of messages is transmitted on the comfort bus. The attacker monitors the bus for the first message of this sequence. Upon detection the attacker immediately sends the sequence to lock the doors. That procedure is sufficient to lock the doors permanently. As the lock is purely electric a manual opening with the key can be overridden as described before.
- **S2 Nuisance** The attacker randomly toggles warning and turning lights, continuously raises the volume of the multimedia system and opens and closes the windows in a random way. This can again be done by inserting CAN messages to the comfort bus.
- **S3 Confinement** The attacker waits on the driver to turn off the car and release the key. Then the central locks are applied and windows are closed followed by a Denial-of-Service attack. This results in locked in passengers as doors and windows are closed an no user input is registered by the ECU.

These three scenarios are implemented in Python using the python-can package for receiving and transmitting messages. A second Raspberry Pi is used for this. In each Scenario the BUS traffic is monitored (O1) and pre-processed (O2).

5.2 Test Results

- **S1 Lockout** When the user tries to open to the car the doors are locked immediately again. That incident alone does not raise any alarm. Subsequent tries to open the car led to an alarm triggered by the detector of the corresponding ID and the complete bus (O4). The Aggregator (O6) fetches data related to the doors from the log file (O2) and passes information to the controller that anomalies were detected on the comfort. It also reports that the doors are probably closed. Since door locks are a concern for security the controller reports to the user over a CLI that unusual behaviour is observed related to the car locks and asks the user if he is trying to open the car. Depending on user input the observer opens the doors.
- **S2 Nuisance** The ransomware used in this scenario has several phases. The ransomware toggles the warning lights **(S21**). This is not sufficient to raise alarm. In the next step the ransomware quickly raises and lowers the volume leading to distracting noise (**S22**). The bus detector (O4) raises alarm due to low entropy values but the ID detector does not. The detected anomaly and information of the log file (O2) are not sufficient for the controller to take any actions. Several runs with different parameters for the ID detector have been performed. Large k values used in the ID detector imply low false-positives rates but does not lead to the detection the attack. Low k values lead to detection but brings high false-positive rates. A productive use is not possible at this moment. However, given the attack is detected the controller turns off the audio system as countermeasure. In the next phase the ransomware raises and lowers the windows in a random way **(S23**). This behaviour carries the sample implications like S22. In total, the current implementation cannot detect the attack reliable without producing high rates of false-positives. In future implementations, multiple analysers, for example ECU Fingerprinting or Artificial Neural Networks, should be used. Moreover, the aggregator could be improved by using a predictor (O5) and more detailed logs (O2), e.g. by building a model of the car that reflects its status.
- **S3 Confinement** The Denial-of-Service attack is detected by ID detectors of windows and locks as well as the bus detector (O4).

Moreover, the Observer reports to the controller that doors and windows are closed. The controller opens the trunk and notifies the user to leave the car through the trunk.

6 Summary and Outlook

This work discusses the possibilities of bringing Organic Computing Observer / Controller architectures into an automotive environment. It is shown that such an adaption is possible and even enhances security and safety of an automotive. The main contribution is the discussion and exploration of necessary adaptation as well as limitation and merits. A first demonstrator and its efficiency in hampering such complex exemplary attack scenarios has been shown. The approach presented in this work allows a reaction on complex threats which single constituent parts would not itself register as a threat. This approach seems well suited for use in other scenarios with similar threat scenarios and basic architecture. In essence all systems which consistent of actuators, sensors, communication bus and computational units carry the same risk of being attacked by injected bus messages. If the system is complex enough that a single injected message does not trigger any alarm or harm by itself while a sequence of injected messages provokes a harmful macro behaviour the same threat scenarios arise. The fact that these system of systems show an observable macro behaviour implies that they influence their surrounding and hence cause inherent safety risk once their security is compromised. One example for such a system would be industrial automation. Industrial automation shares the same basic components like automotive IT and faces the same problems with ever growing complexity. It is without doubt that industrial automation systems are systems of systems. In addition, attacks on these cyber-physical systems also often consist of a sequence of bus messages to the actuators which are each by itself without harm but in their entirety cause a harmful macro behaviour. Examples of malicious attacks on these types of systems might be similar to these presented in here. In the lockout scenario (S1) an attacker would manipulate an industrial system in a way to prohibit its normal function. This might include erratic moving of the industrial robots so a maintenance crew might only be able to approach the robot after powering it down. In either case the functionality is denied to the user and a safety risk arises from the security risk. The possibility of

causing nuisance (S2) by using industrial actuators and HMI is obvious, while a locked in scenario (S3) would require doors to be attached to the industrial system. This mighty only be relevant in risk zones separated by doors and more a topic of smart home automation. However, an adaptive system observing the macro behaviour of such a system again enables the suppression of this threat.

Acknowledgments

The work on organic computing in automotive environments is supported by German Research Foundation project ORCHideas (DFG GZ: 863/4-1). The application to industrial control systems is funded in parts by the German Federal Ministry of Economic Affairs and Energy (BMWi, project no. 1501502B). The authors thank all project staff members and involved students as well as the reviewers for their help.

References & Acknowledgments

[MV15] Miller, C.; Valesek C.: Remote Exploitation of an Unaltered Passenger Vehicle. Black Hat USA

[Su14] Sugimura, T: Junction Blocks Simplify and Decrease Networks When Matched to ECU and Wire Harness. Encyclopedia of Automotive Engineering. 1–7

[Tr98] Trautmann, T.: Grundlagen der Fahrzeugmechatronik: EinepraxisorientierteEinführungfürIngenieure, Physiker und Informatiker, 2009

[Ka08] Kargl,F.; Papadimitratos, P.; Buttyan, L.; Müter, M.; Schoch, E.; Wiedersheim, B.; Thong, T.; Calandriello, G.; di Torino, P.; Held, A.; Kung, A.; Hubaux, J.: Secure Vehicular Communication Systems: Implementation, Performance, and Research Challenges. Communications Magazine, Volume 46(11), 110-118

[Wu08] R.P. Würtz (ed.): Organic Computing. Understanding Complex Systems, doi: 10.1007/978-3-540-77657-4 1, © Springer-Verlag Berlin Heidelberg 2008

[WH05] De Wolf T., Holvoet T. (2005) Emergence Versus Self-Organisation: Different Concepts but Promising When Combined. In: Brueckner S.A., Di MarzoSerugendo G., Karageorgos A., Nagpal R. (eds) Engineering Self-Organising Systems. ESOA 2004. Lecture Notes in Computer Science, vol 3464. Springer, Berlin, Heidelberg

[MSU11] Müller-Schloer, C.; Schmeck, H.; Ungerer, T.: Organic Computing — A Paradigm Shift for Complex Systems, Springer, ISBN: 978-3-0348-0129-4, 2011.

[Al14] Altschaffel, R; Hoppe, T.; Kuhlmann, S.; Dittmann, J.: Towards more Secure Metrics for Assessing the Fitness Level of Smart Cars. Proceedings of the 3rd International Conference on Connected Vehicles & Expo, 149-154

[Ri06] Richter, U.; Mnif, M.; Branke, J.;Christian Muller-Schloer, C.;Schmeck, H.:Towards a generic observer/controllerarchitecture for Organic Computing. GI Jahrestagung (1) 93 (2006): 112-119.

[MWJ+07] Mühl, G., Werner, M., Jaeger, M., Herrmann, K. &Parzyjegla, H. (2007). On the definitions of self-managing and self-organizing systems, KiVS 2007 Workshop: Selbstorganisierende, Adaptive, KontextsensitiveverteilteSysteme (SAKS 2007).

[Bos91] Robert Bosch GmbH, CAN Specification 2.0, 1991, http://www.bosch-semiconductors.de/media/ubk_semiconductors/pdf_1/canliteratur/can2spec.pdf, (last checked: 18.10.2016)

[MN11] Müter, Michael, and NaimAsaj. "Entropy-based anomaly detection for in-vehicle networks." Intelligent Vehicles Symposium (IV), 2011 IEEE. IEEE, 2011.

[CK16] Cho, Kyong-Tak, and Kang G. Shin. "Fingerprinting electronic control units for vehicle intrusion detection." 25th USENIX Security Symposium (USENIX Security 16). USENIX Association, 2016.

[MSGC16] Marchetti, M., Stabili, D., Guido, A., &Colajanni, M. (2016, September). Evaluation of anomaly detection for in-vehicle networks through information-theoretic algorithms. In Research and Technologies for Society and Industry Leveraging a better tomorrow (RTSI), 2016 IEEE 2nd International Forum on (pp. 1-6). IEEE.

[IBM06] COMPUTING, Autonomic, et al. An architectural blueprint for autonomic computing. IBM White Paper, 2006, 31. Jg.