A language-based intrusion detection approach for automotive embedded networks

Ivan Studnia, Eric Alata, Vincent Nicomette, Mohamed Kaâniche, Youssef Laarouchi

A language-based intrusion detection approach for automotive embedded networks

Ivan Studnia[1,2,3], Eric Alata[2,3], Vincent Nicomette[2,3],
Mohamed Kaâniche[2,4], Youssef Laarouchi[1]
[1]Renault S.A.S., 1 Avenue du Golf, F-78288 Guyancourt, France
[2]CNRS, LAAS, 7 Avenue du colonel Roche, F-31400 Toulouse, France
[3]Univ. Toulouse, INSA, LAAS, F-31400 Toulouse, France
[4]Univ. Toulouse, LAAS, F-31400 Toulouse, France
Email: {ivan.studnia, youssef.laarouchi}@renault.com, {ealata,nicomett,kaaniche}@laas.fr

Paper category: Regular paper

***Abstract*—The increase in connectivity and complexity of modern automotive networks presents new opportunities for potential hackers trying to take over a vehicle. To protect the automotive networks from such attacks, security mechanisms, such as firewalls or secure authentication protocols may be included. However, should an attacker succeed in bypassing such measures and gain access to the internal network, these security mechanisms become unable to report about the attacks ensuing such a breach, occurring from the internal network. To complement these preventive security mechanisms, we present a non intrusive network-based intrusion detection approach fit for vehicular networks, such as the widely used CAN. Leveraging the high predictability of embedded automotive systems, we use language theory to elaborate a set of attack signatures derived from behavioural models of the automotive calculators in order to detect a malicious sequence of messages transiting through the internal network.**

Keywords*—*automotive networks, security, intrusion detection, CAN, finite state automata

I. Introduction

The embedding of electronic components into cars is now a well established fact: modern vehicles include up to 100 ECUs [16] (Electronic Control Units, the embedded computers controlling one or more functions of a vehicle). As more and more functions are being managed by ECUs — trending towards fully autonomous cars — the size and complexity of embedded automotive software keep growing [4]. Moreover, a car is now interfaced with other external networks and devices via wired (USB) or wireless (Bluetooth, WiFi, 3G, LTE...) communication. Therefore, the various computing systems embedded in modern cars can no longer be considered as a closed network, increasing the risks of cyber attacks.

Indeed, like any other computing system, an automotive network can be plagued by vulnerabilities, which can be exploited by an attacker connected to it. Such attacks could lead to a privacy breach (for example if the attacker has access to the navigation systems history) or some comfort disturbances (an equipment no longer working properly). More importantly they could also threaten the life of the passengers if safety systems are impacted [15].

While the risks of a cyber attack were previously disregarded by the industry as the ECUs could not easily be accessed from outside a "disconnected" car, the increased risks coming with communication-enabled vehicles [5] have made the implementation of security mechanisms into automotive networks a major topic for manufacturers. This can for example be illustrated by the numerous European projects such as SEVECOM [9], PRESERVE [1], EVITA [7] or OVERSEE [6].

As the typical lifetime of an automobile is about 20 years, the use of purely preventive measures such as firewalls or misuse-based IDS may not be sufficient since the initial rules may become outdated. In this context, we believe that such security measures should be complemented by anomaly detection techniques that provide the ability to address previously unknown attack scenarios that could target modern vehicles. Compared to traditional computer networks, the interactions between ECUs are strictly defined, the protocols are currently more simple, and the possible user inputs (at least concerning the safety-critical aspects) are limited. This results in a relatively high predictability of the network reaction given its current state. Thus, by defining a language characterizing the messages exchanged through the network, one could detect if a message sent on the network is legitimate or inappropriate.

In this paper, we introduce a network-based intrusion detection system (IDS) fit for a typical embedded vehicular network. More specifically, we present a method leveraging the specificities of the automotive network in order to generate a set of forbidden message sequences which can be used to perform intrusion detection. This paper is organised as follows. We first give an overview of the related work addressing automotive security in section II. In Section III we present our proposed IDS for in-vehicle networks, its design and formalisation. We illustrate it with an example in Section IV and present a possible optimization to reduce the spatial complexity in Section V. The performances of our approach are then assessed in Section VI. Section VII is devoted to discussions about the limitations of our approach. Finally, Section VIII concludes this paper.

II. Related Work

Automotive networks consist of several domains and securing such networks requires a holistic approach, where security is enforced in every domain. While solutions for securing the infrastructure and the communication interfaces between the vehicle and the outside world exist, they are beyond the scope of this paper. Concerning the embedded automotive network, security solutions can be grouped in three main categories: 1) Cryptographic techniques to authenticate or encrypt the packets on a bus; 2) Solutions to ensure integrity of the embedded software and 3) Solutions to detect anomalies occurring in the network. More information can for example be found in [22], [10] or [21].

For desktop computers, intrusion detection systems are now common and comprehensive surveys of such systems can be found in the literature (see for example [13]). However, in the automotive world, we could only find a few examples of intrusion detection systems and to the best of our knowledge, no complete implementation of such a system has been documented.

A specification-based IDS for automotive networks was introduced in [12]. Their system works by pairing each ECU with a detector that checks if the frames sent or consumed by the related ECU are compliant with a set of rules based on specifications of both the protocol and the ECUs. Some attacks require a communication between detectors to be detected. Moreover, in the method presented in [14], intrusion detection is performed by every node on the network, which tries to detect if other nodes are impersonating it by checking the frames headers. While such approaches show interesting results, they require some modifications of the software embedded in the ECUs. However, new vehicles are seldom designed from scratch and numerous components are reused from previous projects or COTS[1] products that cannot easily be altered.

Müter et al. proposed two possible approaches for network-based anomaly detection. First, [18] describes a set of sensors to perform intrusion detection in an automotive network without theoretically raising any false positive. [17] presents a statistical approach in which an alert is raised when sudden variations of the entropy are witnessed. The notion of a vehicular state (parked, driving forward or backwards...), which could be modeled by a finite state machine and determine what value of entropy should be used as the baseline, is mentioned but not used in their experiments.

If a well designed signature-based IDS raises very few false positives (or even none), it requires regular updates to maintain its signature base up to date. On the other hand, anomaly-based intrusion detection may be able to detect previously unknown attack patterns, but the high complexity of an automotive network makes it difficult to design a precise enough statistical model to prevent false positives while still allowing exceptional but perfectly legitimate situations.

[1]Commercial Off-The-Shelf

Our work expands on these previous approaches as our goal is to implement and test an IDS that leverages the specificities of an automotive network to perform its tasks. More specifically, as we consider a manufacturer's point of view, we want to use our advanced knowledge of the embedded network and the behaviour of the interconnected ECUs in order to be able to infer the state of its nodes at any time. This then allows to detect ongoing attacks by checking whether a frame belongs to an attack sequence with regards to the currently recorded state of the system. As our IDS is derived from the ECUs specifications (assuming these specifications are correct), we do not have to rely on statistical analysis to create our models, therefore avoiding potential false positives.

III. Designing an automotive intrusion detection system

The automotive IDS that we propose in this paper aims at detecting ongoing attacks on the internal network based on data gathered from the network traffic. This section is devoted to the detailed presentation of this IDS: subsection III-A describes the attack hypotheses, III-B explains our approach to detect these attacks, III-C sums up our solution and III-D gives the formal details of our implementation.

A. Attack hypotheses

In 2011, Checkoway et al. [5] have proven that it is possible for an attacker to remotely gain an access to the embedded network, as they took control over ECUs acting as communication interfaces. They then successfully managed to reproduce the attacks described in [11], effectively remotely compromising a car from short communication range (via Bluetooth or by indirect physical access scenarios where the legitimate user unwillingly connects an infected device to his car) to very long communication range (3G).

In our work, we assume that an attacker already gained the ability to remotely send messages to at least one internal bus but is not able to physically tamper with the internal network (as this could allow him to completely bypass any network-based security device). In other words, our work focuses on detecting a successful breach into a car's embedded network but not on the possible causes of this breach (remote attack, sabotage...). We also assume that the attacker may possess advanced knowledge (such as the network architecture or the content of specific CAN frames) about the automotive network he is attacking.

From a network perspective, it is assumed that the attacks correspond to the transmission through the network of a frame or a sequence of frames sent in order to divert the car from its standard behaviour. With this in mind, the attacks we want to detect fall into the following categories:

1) Emitting frames that do not conform to the specifications (unknown identifier, data field of the wrong size, incorrect CRC...).
2) Sending periodic, forged frames while the legitimate corresponding frames are still being sent by the system (for example, malicious frames

requesting to turn off the beams while they are legitimately being switched on).

3) Replacing legitimate frames by malicious ones; the corresponding legitimate frames are no longer seen on the network.
4) Forging event-related frames, that are only sent asynchronously in reaction to a given event instead of being sent periodically with an updated content.

To the best of our knowledge, most approaches developed for automotive applications address the first two categories of attacks and do not specifically focus on the latter.

B. Proposed approach

Our goal is to develop a (relatively) cheap system which can be deployed on current architectures and subsequently ported to future models with a low amount of efforts. To do so, we chose not to alter any already existing node of the network. Hence, intrusion detection is based only on the information gathered from the messages observed on the buses our system has access to.

Among the four kinds of attacks described above, the first two cases are easy to detect: 1) Erroneous frames can be detected by checking a message against a set of specifications, and 2) Periodic frames sent in addition to legitimate traffic can be detected by an increase in the frequency of the targeted frames. Attacks falling into these categories are detected by all the IDS presented in Section II. Indeed, such detections are easy to implement through a set of formal rules. In such cases, analyses on a single frame are sufficient to detect an anomaly. However, in order to detect attacks falling within the last two categories, taking only one frame into account may not be sufficient as it may by itself be compliant to every formal rule. Instead, the key to detect these attacks relies on the correlation of contextual information about the monitored frames. These two aspects (rules and correlation mechanisms) of detection are considered in [18] via the description of 8 categories of sensors detecting different kinds of anomalies. Some of these sensors check the frames for consistency with the system specifications while others perform consistency checks between the content of several frames. However, no implementation of the sensors is presented. Besides, [17] and [16] present approaches where the system has to learn the standard network behaviour. While we consider that such approaches are performing some correlation, this correlation is done with respect to what has been observed during the learning phase, and not to what has happened during the actual monitoring. Moreover, such methods are more prone to false positives since it is hard to guarantee that all possible legitimate use cases were taken into account.

In traditional information systems and networks, IDS must address the fact that they may not precisely know every node of the monitored system, the whole range of their behaviour (as people are using them) nor the entire configuration of a network as nodes could be added or removed very often. This results in a wide (possibly infinite) range of possible rules and/or configurations to be defined if one wants to be able to monitor everything. This is not the case in a car: the embedded network architecture usually does not change during the lifetime of the vehicle and there are not many different possible human inputs. This stability can be used to our advantage as the behaviour of a (sub)system of ECUs is therefore quite predictable. The car manufacturers have an advanced knowledge of the system and can use the ECUs specifications as a reference point to define the expected behaviours of the systems they want to monitor.

However, directly confronting models derived from the specifications to the actual traffic requires a good synchronization between the monitoring system and the nodes of the network. Indeed, we may need to keep monitoring the vehicle after a first detection (for example, if the detected message sequence does not target a safety-critical system, we could consider that it is not necessary to immediately alert the driver or stop the vehicle). As our system just witnessed a deviation from the expected behaviour, it can no longer be sure of the state of the monitored systems. One solution to this issue is to give the system the ability to communicate with each ECU in order to ask them their current state, which adds a lot more complexity to its implementation and would defeat our objective of not altering the already existing nodes. Therefore, we need to find a detection method that does not require such a perfect synchronization with the nodes.

Our idea is to base our detection on a list of forbidden sequences (i.e. signatures) derived from such aforementioned models. The main advantage of this method is that the analysis may resume after a de-synchronization (although it may need to drop the current sequence and start over with the next frame). Moreover, the deterministic aspect of the automotive network can potentially allow us to create a comprehensive signature-base which could last for the lifetime of the car without requiring further updates (as long as no node is replaced nor added into the network or the specification of some nodes is updated).

In order to perform a correlation between the monitored frames, we must gather and use data from previously seen messages to infer the current state of the car (or at least the monitored subsystem), which will allow us to determine if a frame is consistent with the current context. Accordingly, the information needed is:

- Similar data from distinct sources (for example speed-related data can come from the wheels as well as from the navigation system). Some data can also be emitted in several distinct frames for safety reasons. This redundancy could also be used for security. This data may come from different subnetworks.
- An history of the previous frames emitted by the monitored subsystem.

Therefore, two levels of detection can be performed for each frame: first formal checks to know whether the frame is compliant with the specifications, then consistency checks to detect if the frame is consistent with the current state of the system.

C. IDS Overview

In this subsection, we describe the different features of our IDS, with a specific focus on the context-sensitive anomaly detection.

1) Location: In order to retrieve information from distinct sources, our IDS needs to be connected to all the monitored networks. Indeed, a car's embedded network usually consists of two or more subnetworks of ECUs linked together via some ECUs acting as gateways. During an attack, such gateways become prime targets as they would allow an attacker to reach another part of the network. Therefore, our IDS can either be included in one of the existing gateways or become a new node connected to all the buses to be monitored (however this could create a link between otherwise separated networks, potentially increasing the risks). A distributed IDS, with several nodes on distinct subnetworks would require the design of a communication channel between the different components. At the time of writing this paper, we have not tested such a solution.

Let us consider a possible modern automotive network architecture such as the one shown in Figure 1, based on an example of the EVITA project [2]. Possible locations for the IDS could be: A) on a bus shared by several gateways, B) into an already existing gateway ECU or C) several smaller modules distributed across the network.

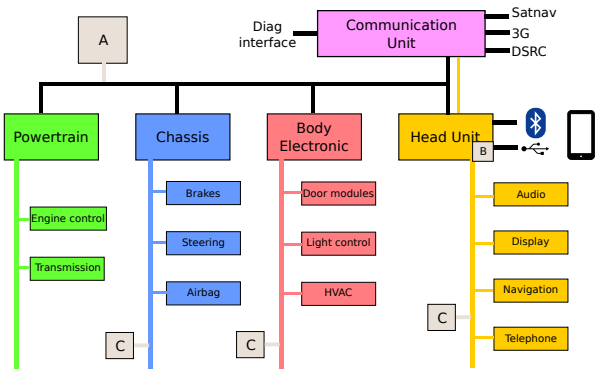


Figure 1: Possible IDS implementations in a modern automotive network architecture

2) Principle: As seen in III-B, some anomalies can be detected by checking a single frame while others require to keep track of some of the traffic history. As we want to be able to detect both, we need to perform two successive analyses. Those are summed up in Figure 2. First, a preprocessing is required for identifying the frame type (in the case of CAN, it is done with its ID field) and therefore preparing the content of its data field for further analyses. The first part of the detection consists in checking whether a frame complies with a set of rules defined by the protocol used and/or the manufacturer's specifications. When a rule is violated, an alert may be raised according to the severity and frequency of the violation (depending on the affected system). As this is not the main focus of this paper, we will not get into further details for this part. An example of such rule-based system can be found in [19]. Finally, the context-related analysis is done in a signature-based intrusion detection fashion, through the use of finite state automata. The signatures generation and their processing are detailed in the following section.

D. Formalisation

In this section, we describe the proposed approach for signature generation. In our context, a system is composed of several ECUs involved in carrying a specific task or set of tasks. An ECU may contribute to several functions. For example, the system "Braking" would include the ECUs monitoring the pedals, the handbrake, the brakes and the (brake) lights. As such systems are often specified via state machines, we use formal language theory as a convenient way to describe them. For each ECU, we define their language L_{ECU} as follows:

- The symbols are the different requests emitted onto or received from the network, restricted to what is relevant to the system under consideration. An alphabet Σ_{ECU} is the set of all these symbols.
- The words are all the valid sequences of symbols, i.e., sequences that can happen during a legitimate use of the system.

We can then create a finite state automaton (FSA) describing this language. It represents a model of the ECU behaviour as perceived from the network.
We remind that a FSA consists of a quintuple $(Q, \Sigma, \delta, I, F)$ where

- Q is the finite set of states.
- Σ is the finite alphabet.
- $\delta : Q \times \Sigma \to P(Q)$, where $P(Q)$ is a subset of Q, is the transition function
- I is the set of start states.
- $F \subseteq Q$ is the set of final (or accept) states.

A deterministic finite automaton (DFA) is a FSA where

- There is only one start state: $I = \{q_0\}$
- The transition function is deterministic: for a given current state and a given symbol, there is one and only one state to be reached.

In the rest of this paper, all automata are considered complete, which means that for each state, there is an outgoing transition for every symbol of their alphabet. States from which there is no sequence eventually leading to a final state are called dead states. All such states can be grouped into one unique dead state.

The language of the system denoted as L_{sys}, and its corresponding FSA denoted as A_{sys}, are defined as the synchronized parallel composition of all the automata of its ECUs. Their alphabet is denoted as Σ_{sys}.

The synchronized parallel composition of two complete DFA A and B is defined as:

$$A \parallel B = (Q_A \times Q_B, \Sigma_A \cup \Sigma_B, \delta, I_A \times I_B, F_A \times F_B)$$

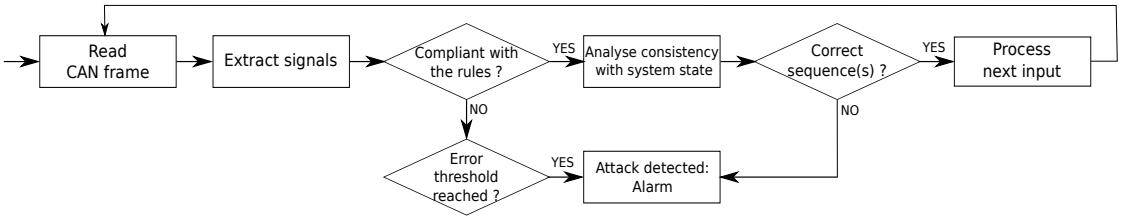


Figure 2: Flowchart of the intrusion detection process

where

$$\delta(q_A.q_B, s) = \begin{cases} \delta(q_A, s).\delta(q_B, s) \text{ if } s \in \Sigma_A \cap \Sigma_B \\ \delta(q_A, s).q_B \text{ if } s \in \Sigma_A \wedge s \notin \Sigma_B \\ q_A.\delta(q_B, s) \text{ if } s \notin \Sigma_A \wedge s \in \Sigma_B \\ q_A.q_B \text{ otherwise} \end{cases}$$

We remind that our goal is to detect an attack on the network by checking whether the frame emitted on a bus is consistent with the system behaviour history. In other words, we want to detect when a frame belongs to a sequence that is not part of any valid sequence of L_{sys}. An invalid (or forbidden) sequence can be defined as a sequence that is valid up to a certain point where an unexpected symbol eventually appears. Formally, if w is an invalid word for L_{sys} and $w = s_0 s_1..s_n$ where all s_i are symbols from Σ_{sys}, then:

$$\exists v \in L_{sys}, \exists k \in [0, n-1], s_0..s_k \in pref(v) \wedge s_0..s_{k+1} \notin L_{sys}$$

where $pref(v)$ is the set of all the prefixes of v. To do so, we will derive a set of such forbidden sequences from L_{sys}.

In the following, we describe the different steps of the method that we adopted in order to obtain the set of invalid sequences:

1) Starting from L_{sys}, we construct the set containing all the words that start as words of L_{sys} (i.e., that belong to $Pref(L_{sys})$, the set of the prefixes of all the words of L_{sys}) concatenated with an extra symbol from Σ_{sys}.
2) Let $\overline{L_{sys}}$ be the complement of L_{sys}. $\overline{L_{sys}}$ thus contains every word that is not in L_{sys} and therefore every possible invalid sequence of requests.
3) By computing the intersection of the sets described in steps 1 and 2, we therefore obtain the set

   $$\{w \in \overline{L_{sys}} | \exists (x, s) \in Pref(L_{sys}) \times \Sigma_{sys}, w = x.s\}$$

   This set corresponds to $(Pref(L_{sys}).\Sigma_{sys}) \cap \overline{L_{sys}}$ and contains all the invalid sequences truncated after their first deviation from a normal behaviour. At that point, we have a language that could theoretically be used to detect any deviation from the normal behaviour as long as it is perfectly synchronized with the monitored system.
4) However, as we want to be able to restart the observation after a detection (without having to restart the whole car), we must be able to detect attacks starting at any time of a sequence, that is to say the suffixes of an attack sequence. The corresponding sequences are included in $Suf(Pref(L_{sys}).\Sigma_{sys} \cap \overline{L_{sys}})$, where $Suf(L)$ is the set of all the suffixes of a language L.
5) However, this set also contains words that are portions of valid sequences of L_{sys} (i.e., they are part of the set of factors of L_{sys}, or $Fact(L_{sys})$). Raising an alert for such words could be a false positive. We therefore remove from this set all words belonging to $Fact(L_{sys})$, which gives us the following set:

   $$Suf(Pref(L_{sys}).\Sigma_{sys} \cap \overline{L_{sys}}) \cap \overline{Fact(L_{sys})}$$

That last set, called $S_{attacks}$, describes the language of the attacks carried over an automotive network for a given (sub)system. $S_{attacks}$ is created from the rational language L_{sys} and FSAs are closed under all the operations used to perform this generation. $S_{attacks}$ is therefore also a rational language $L_{attacks}$.

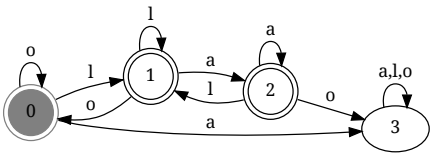
IV. Example

In all the following figures, grayed states are the start states of an automaton and double-circled states represent final states.

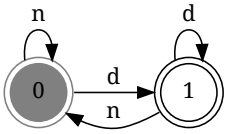
Let us consider the automata shown in Figure 3 representing 3 fictional ECUs involved in the control of the low beams. The transitions correspond to signals sent or received through the network (as there is no source nor destination authentication on CAN, there is no difference between these two cases from an observer point of view). The "command" ECU sends instructions for the ECU controlling the lights: turning the beams on (l), off (o) or activating the automatic mode (a), which relies on the data sent by the light sensor: day (d) or night (n). When receiving l or o instructions, the lights controller simply executes them (and ignores data coming from the light sensor). However, when asked to switch to automatic mode, it will send a frame telling whether it chose to turn the lights on (a_l) or off (a_o). As we consider that the observation can be stopped at any time, we represent those ECUs with prefix-closed automata.

The minimized result of their synchronized parallel composition is given in Figure 4, this automaton corresponds to the language L_{sys}.

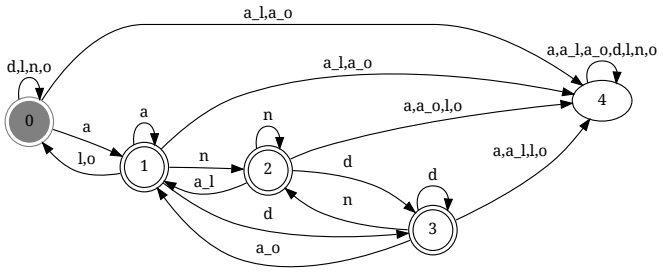
To illustrate the composition, let us consider for example that the 3 ECUs represented by automata (a), (b) and (c) are in their initial state. When the symbole o is read, automata (a) and (c) take a reflexive transition toward their respective initial states. The language of automaton (b) does not contain o, therefore, (b) stays in



(a) Command



(b) Light sensor



(c) Light controller

Figure 3: Initial ECU automata

its current state, which is the initial state. Therefore, on their synchronized parallel composition, symbol o labels a reflexive transition on its initial state. However, if the symbol l is read instead, automaton (a) takes a transition to state 1 and automaton (c) takes a reflexive transition to state 0. Here also, the language of automaton (b) does not contain l, therefore, (b) stays in the initial state. Therefore, as at least one of these automata is no longer in its initial state, the symbol l labels a transition from the initial towards another state on their synchronized parallel composition.

The automaton $A_{attacks}$ recognizing $L_{attacks}$ is shown in Figure 5. A forbidden sequence is detected when the final state is reached. For example, we highlighted (with dotted lines) the transitions corresponding to the forbidden sequence *a,n,a_o* (automatic mode is on, the light sensor says it is night time and yet the lights controller turns the lights off).

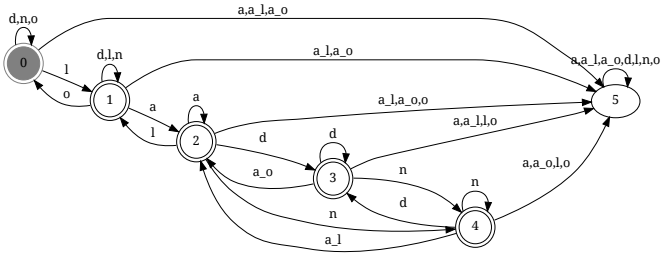


Figure 4: Parallel composition of automata (a), (b) and (c)

V. Scalability

In an automotive context, the processing of a frame has to be as short as possible as a too long detection delay could have serious consequences. Moreover, saving memory space on an embedded software can reduce the costs of the corresponding ECU. Both these constraints can be expressed in terms of temporal and spatial complexity.

First, a naive implementation could consist in creating a DFA $A_{attacks}$ that corresponds to the language $L_{attacks}$, as DFAs allow fast signature-matching performances. Indeed, time complexity is not an issue while processing a DFA since there is only one possibility for each (state, transition) couple. The processing can thus be done in linear time. For example, a simple way to represent a finite state automaton is by using a two-dimensional array where each line is a possible state and each column represents a possible transition. We estimate the spatial complexity by calculating the maximum size (i.e., the number of states) of the automata generated by the previously described method. Complexities in the worst-case for the required operations on finite state automata are given in table I. We base our calculations on results that can be found in the literature [23], [3]. Therefore, if we suppose that the initial language L_{sys} can be represented by a complete DFA containing n states, then

1) $Pref(L_{sys})$ state complexity is n. Practically, it is the same automaton as L_{sys} where every state leading to a final state is made final too.
2) $Pref(L_{sys})$ being by definition prefix closed, $Pref(L_{sys}).\Sigma_{sys}$ state complexity is $n+2$ in the worst case, as a DFA accepting $(Pref(L_{sys}).\Sigma_{sys})$ can be constructed from a minimal DFA accepting $Pref(L_{sys})$ and adding at most two states. First, we create a new start state with outgoing transitions having the same targets than those of the original start state but with no incoming transition (the original start state thus loses its status). Then, we remove the reflexive transitions on the former dead state and make it a final state. A new dead state, reachable only from the former dead state is then added.
3) State complexity of $\overline{L_{sys}}$ is n.
4) Therefore, in the worst case, the state complexity of $(Pref(L_{sys}).\Sigma_{sys}) \cap \overline{L_{sys}}$ should be $O(n^2)$. However, as both sides of the intersection come from the same language L_{sys}, they share many similarities. Indeed, by applying a transformation to the $\overline{L_{sys}}$ automaton similar to what is described in step 2), but without altering its final states, we obtain an automaton that still accepts $\overline{L_{sys}}$ but has the same structure as the one described in 2). Therefore, an automaton with the same structure where the final states correspond to the states that are final in both these previous automata accepts the intersection of their languages and its state complexity is $n+2$.
5) There is a simple way to construct the automaton describing the suffix closure of a regular language by taking a DFA recognizing this language and turning every state leading to a final state into an initial state. However, this makes the automaton a Nondeterministic Finite Automaton (NFA). Creating a DFA recognizing the same language as an

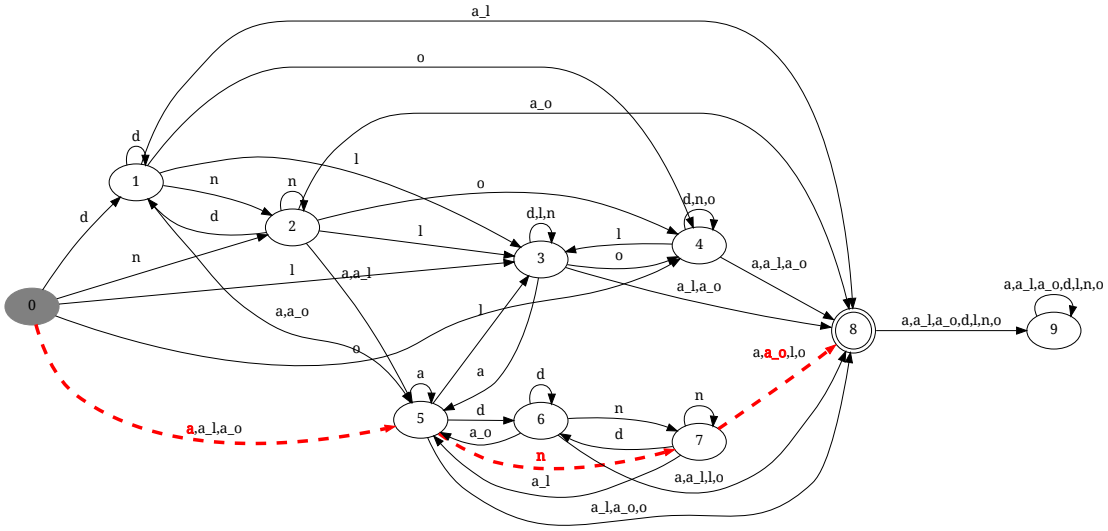


Figure 5: Automaton $A_{attacks}$

NFA is possible (via the powerset construction). However, in the worst case, the resulting DFA possesses 2^n states.

6) The same goes with the Factor closure of L_{sys} as $Fact(L_{sys}) = Suf(Pref(L_{sys}))$. Here also, a DFA recognizing $Fact(L_{sys})$ may also have 2^n states in the worst case.

Therefore, constructing a DFA $A_{attacks}$ recognizing the language has exponentially more states than the original automaton A_{sys} describing the system. This becomes problematic for complex systems with potentially hundreds of possible states. In order to mitigate this potential issue,

Table I: State complexity of basic operations on finite state automata

Automaton	Size (worst case)
L_1	m
$\overline{L_2}$	n
$\overline{L_1}$	m
$L_1 \cap L_2$	mn
$L_1 \cup L_2$	mn
$Pref(L_1)$	m
$Suf(L_1)$	m (NFA)
$DFA(L_1)$ if L_1 NFA	2^m

we propose the following solution:

Let $L_{left} = Suf(Pref(L_{sys}).\Sigma_{sys} \cap \overline{L_{sys}})$ and $L_{right} = \overline{Fact(L_{sys})}$. We now have $L_{attacks} = L_{left} \cap L_{right}$.

By definition, L_{right} contains all the words on Σ that are not factors of L_{sys}. Its intersection with L_{left} ensures that we stop (and/or restart) the observation after the first deviation from a potentially legitimate sequence has been detected. Therefore, we can detect words from $L_{attacks}$ by just considering an automaton accepting L_{right} and stopping the observation the first time the final state is reached. However, as $L_{right} = \overline{Fact(L_{sys})}$, using a DFA for this would raise the same issues as before. Instead, let us use the fact that $Fact(L_{sys}) = Suf(Pref(L_{sys}))$.

Let A_{right} be a DFA that accept the language $Pref(L_{sys})$. The size of A_{right} is $O(n)$. Manipulating A_{right} as is would still require a strong synchronization with the monitored system. This is why we introduce the set S_{right} which initially contains all the states of A_{right}. This set represents all the possible states in which the monitored system could be at the time of observation. Whenever a new relevant signal is processed, a new set is created: it contains the states that can be reached from the states contained in S_{right} by taking the transition corresponding to that signal.S_{right} then becomes this new set for the next step. This method lets us browse A_{right} as if all its states were initial states.

The **delta**(A, s, x) procedure in Algorithm 1 returns the state reached after activating the transition corresponding to symbol x from state s in automaton A. The **isFinal**(s) procedure returns $True$ if state s is final. Algorithm 1 describes the procedure used to perform the detection using these two lists when a symbol x is read. It works as follows.

1) Whenever an incoming frame contains data that corresponds to a symbol of Σ_{sys}, all the elements of S_{right} are updated with respect to the transition function of A_{right}.
2) S_{right} corresponds to the possible evolutions in $A_{right} = Pref(L_{sys})$, although we are actually interested by the elements of $\overline{Fact(L_{sys})} = \overline{Suf(Pref(L_{sys}))}$. Therefore, the observed sequence is in $\overline{Fact(L_{sys})}$ iff it is not a suffix of $Pref(L_{sys})$, i.e., iff no element of S_{right} corresponds to a final state of A_{right}.
3) If such a condition is met (i.e., all the states of S_{right} are not final), then the observed sequence is L_{right}, and therefore in L_{attack}. Else, we continue the observation.

Therefore, we are able to determine whether a sequence is a word of $L_{attacks}$ without having to build the DFA $A_{attacks}$. The memory requirements corresponding to the execution of this method can be estimated as follows:

Algorithm 1 Intrusion detection

```
1: procedure DETECT(S_right, x)
2:     counter ← 0
3:     for i ← 0, size(S_right) − 1 do
4:         S_right[i] ← delta(A_right, S_right[i], x)
5:         if ¬isFinal(S_right[i]) then
6:             counter ← counter + 1
7:         end if
8:     end for
9:     detection ← (counter2 = size(S_right))
10:    return detection
11: end procedure
```

- $n \times |\Sigma_{sys}|$ for A_{right}
- n for S_{right}

Which makes a total of

$$n(|\Sigma_{sys}| + 1)$$

The tradeoff in temporal complexity is that for each step, we execute the delta function at most n times for A_{right} (i.e., once for each element of S_{right}) instead of just once if we were using a deterministic version of $A_{attacks}$.

Figure 6 describes a run of Algorithm 1 using the example system from section IV:

1) First, S_{right} is initialized with all the states of A_{right}.
2) Symbol 'a' is read: For every state in S_{right}, we retrieve the corresponding state reached when taking the transition corresponding to 'a' from them on the automaton A_{right}, equivalent here to the A_{sys} of Figure 4 as A_{sys} is prefix-closed. A new set is then created, becoming S_{right} for the next step.
3) As long as the requirement described in Algorithm 1 is not met, we continue the observation and read the next input (here, 'n').
4) However, when symbol 'a_o' is read, we are in a situation where all the states of S_{right} are not final (the red cases). In such a case, this means that the sequence 'a,n,a_o' is a forbidden one: an alert is raised.

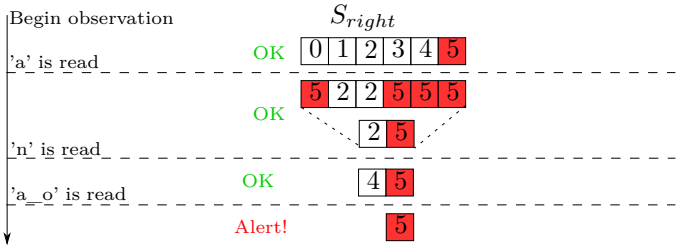


Figure 6: A run of Algorithm 1 for the sequence a,n,a_o

In order to estimate the efficiency of this solution in the case of large systems, we randomly generated DFAs of different sizes (although with an alphabet size arbitrarily fixed at 20 symbols). Each automaton is then used as an A_{sys} from which we derived the corresponding A_{right}. For each sample, we then randomly generated a sequence that was designed to raise an alert on the tenth symbol and have it analysed with our method. Before each step (i.e., whenever a new symbol is analysed), we counted the number of elements of S_{right}. Figure 7 shows the results of this experiment for minimal automata A_{sys} consisting of 10, 100 or 1000 states (resp. the red, green and blue curves). We can see that the size of S_{right} decreases exponentially with each step. This means that even if this method may be too slow to detect an attack on time against a complex system happening right after the IDS started, a quick detection can be performed after a few symbols have been processed, since the size of S_{right} will have drastically reduced.

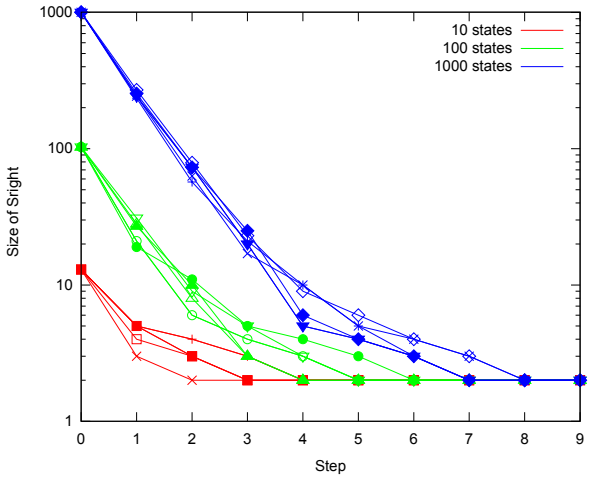


Figure 7: Evolution of the size of S_{right} for different sizes of A_{sys}

VI. Evaluation

Testing the proposed IDS on a real vehicle and under representative traffic conditions and attack scenarios is a very challenging task. At this stage of our work, we assessed the performances of our method based on logs of the CAN network traffic of a car that has been recorded under "standard" conditions (i.e., without the injection of malicious messages). In order to simulate some attacks, we modified these logs by adding frames (for the cases where an attacker inserts additional traffic in the network) or modifying existing frames (for the cases where the attacks consist in altering the messages sent by one or several ECUs). We then used the altered logs to test our intrusion detection system.

The tests were performed on an Intel *Core i7-3720QM 2.60GHz* processor. In order to get as close as possible to an actual automotive hardware, we used only one core clocked at 1.2Ghz. The early prototype used for the tests has been developed in Python. The values presented in this section are therefore not representative of a fully optimised implementation of our method.

For every case presented afterwards, the results were obtained by analysing 100 times a log that contained

310440 frames, corresponding to 205s of actual CAN traffic. On average, this means we need to analyse 1514 frames per second. We used two systems A and B, which we cannot describe here for confidentiality reasons, to perform the tests. The first one can be represented with an automaton containing 21 states while the second one possesses 108 states. For system A, we obtained the performances measured during the automata based checks are given in Table II. In this table, we confronted the two methods described in this paper. The first row shows the timings measured by computing and using the DFA $A_{attacks}$ (which contains 54 states) to perform the detection. The second row shows the timings measured when using the DFA A_{right} and Algorithm 1 introduced in section V.

As mentioned in Section III, using the DFA $A_{attacks}$ allows to make the automata based checks in constant time, as evidenced by the mean time and the minimum time being equivalent. The maximum observed time actually corresponds to the cases where an anomaly was detected. On the other hand, the method using A_{right} does have a smaller automaton but takes more time.

The monitored CAN traffic corresponds to an average of 1514 frames per second, or $660\mu s$ between each frame. Therefore, for simple systems, we are able to perform an analysis in real time with both methods since the automata based checks take $176\mu s$ in the worst case.

Table II: System A – Average, minimum and maximum durations observed during the analysis of 310440 frames

Method	Number of states	Average duration	Minimum duration	Maximum duration
$A_{attacks}$	54	$12\mu s$	$10\mu s$	46 μs
A_{right}	21	$38\mu s$	$33\mu s$	176 μs

Concerning system B, the corresponding automaton $A_{attacks}$ possesses 366 states. As the analyses based on this method are done in constant time, the corresponding results are equivalent to those obtained for sytem A. Therefore, in Table III, we only included the results obtained using A_{right} and Algorithm 1. In this case, while the minimum and average durations are still satisfactory, the maximum is above the threshold of $660\mu s$ and can therefore cause an occasional delay between the monitoring and the actual traffic. The worst case with this method corresponds to the beginning of an analysis, where the set S_{right} contains all the states of A_{right}. However, as we saw in Section V, the size of S_{right} significantly decreases with each frame analysed. This fact can be observed here with an average duration of $49\mu s$ that is significantly lower than the maximum of $779\mu s$. The major part of the analyses can therefore be performed in real time in this case too.

Table III: System B – Average, minimum and maximum durations observed during the analysis of 310440 frames

Method	Number of states	Average	Min	Max
A_{right}	108	$49\mu s$	$20\mu s$	779 μs

As a result, before modelling all the ECUs and systems of an embedded network, it is important to precisely establish which systems need to be synchronized together. Indeed, if two large systems are not correlated at all, it may become more advantageous to carry out two separate analyses with two smaller automata rather than a single analysis using a more complex one. If the use of a DFA $A_{attacks}$ is no longer possible, the latter situation would require a much bigger S_{right} set and would therefore lengthen the analysis duration in the worst cases.

VII. Discussions

In this paper, we focused on the conceptual approach of an automotive, specification-based, IDS. Using examples from an automotive manufacturer, we have been able to test this method and successfully generate attack signatures derived from ECUs models. The implementation of this IDS for an actual automotive network together with the evaluation of its efficiency in a real life scenario is the subject of ongoing work. This section is devoted to a critical discussion about the limitations of our solution concerning the attacks it can or cannot detect.

First, the automata used by our solution are based on the specifications of the network and of the ECUs. However, it can happen that parts of the actual system do not strictly adhere to these specifications (as witnessed in [11]) and some small deviations can occur. In such cases, our system might consider that there is an attack happening although it is actually observing one of such deviations. A way to circumvent this issue could be to generate (or assist the generation of) the A_{sys} automata via machine learning. However, this brings the usual issues coming with statistical anomaly detection mentioned in Section II.

Moreover, some attacks may never be detected if we miss some of their initial frames. For example, let us consider two sequences. The first one, *a(xyz)*o*, is legitimate while the second, *b(xyz)*o*, is forbidden and the signals corresponding to *a* and *b* are not sent periodically. In this case, if our system does not register the signal corresponding to the first symbols, it will not be able to detect the attack corresponding to the second sequence. Therefore, if an attacker is able to conveniently force the IDS to restart (for example if the IDS is integrated into another ECU that can be restarted via a specific CAN frame), he may be able to hide the attack from it.

Finally, in this paper, we are only considering "basic" FSAs. While these are convenient as they are easy to manipulate for initial tests, their limited expressiveness can hinder the modeling of more complex systems. For example, the time does not directly appear in our automata, as we instead chose to include the timing between periodic frames into the formal rules constituting the first step of the detection (see Figure 2) and just focus the second step on the order in which the frames are emitted. However, there may be some attacks where not only the order but also the timing between (consecutive) distinct frames could matter. At this stage, our IDS does not detect such attacks. An interesting evolution will be to use more expressive automata (see for example [20]) in order to improve the detection capacities and the efficiency of our system.

VIII. Conclusion

In this paper, we presented a solution for generating a language that can be used for performing intrusion detection in an embedded automotive network. Using the fact that automotive networks are designed to operate according to a strict set of rules, we used formal language theory to derive a language characterizing the attacks we wanted to detect from the network and ECU specifications. We also described an algorithm that allows to overcome the potentially excessive memory usage due to an exponential increase in states that this language could cause albeit requiring a more intensive CPU usage. However, tests of this method on randomly generated automata showed that the extra required operations decrease exponentially as the monitoring progresses. We presented an evaluation of the performances of the methods presented in this paper on an early prototype using simulated attacks performed on logs of an actual CAN network. We are currently working on an instantiation of our solution on a real automobile and on implementing the proposed IDS onto an ECU-scaled processor.

Obviously, performing intrusion detection is only a first step. So far, we focused exclusively on the detection of intrusions on an automotive network. However, another critical problem that is out of the scope of this paper is how to react when an intrusion is detected. Different possibilities could be distinguished: 1) logging the detected attack sequence for forensics analysis, 2) developing automatic recovery actions that are consistent with safety rules, and 3) alerting the driver. The question of how to react in an automotive environment once an intrusion is detected is an important topic which requires further development. Indeed, one cannot disturb the driver if the detected intrusion is not likely to have a serious safety impact. Perhaps several levels of alerts could be used [8], depending on the criticality of the system being targeted by an attack.

Moreover, in-vehicle network architectures are continuously evolving and can comprise many subnetworks. As these architectures gain in complexity, a distributed IDS with smaller modules being installed in each of the concerned subnetworks could prove to be an interesting solution, both for logistic and cost reasons.

References

[1] About PRESERVE. http://www.preserve-project.eu/about (2011), [Online; accessed February-2013]

[2] Apvrille, L., El Khayari, R., Henniger, O., Roudier, Y., Schweppe, H., Seudié, H., Weyl, B., Wolf, M.: Secure automotive on-board electronics network architecture. In: FISITA World Automotive Congress, Budapest, Hungary. vol. 8 (2010)

[3] Brzozowski, J., Jirásková, G., Zou, C.: Quotient complexity of closed languages. In: Computer Science–Theory and Applications, pp. 84–95. Springer (2010)

[4] Charette, R.N.: This car runs on code. IEEE Spectr. 46(3), 3 (2009)

[5] Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S., Koscher, K., Czeskis, A., Roesner, F., Kohno, T., et al.: Comprehensive experimental analyses of automotive attack surfaces. In: Proc. 20th USENIX Security. San Francisco, CA (2011)

[6] Groll, A., Holle, J., Ruland, C., Wolf, M., Wollinger, T., Zweers, F.: Oversee a secure and open communication and runtime platform for innovative automotive applications. In: 7th Embedded Security in Cars Conf. (ESCAR). Düsseldorf, Germany (2009)

[7] Henniger, O., Ruddle, A., Seudié, H., Weyl, B., Wolf, M., Wollinger, T.: Securing vehicular on-board it systems: The EVITA project. In: 25th VDI/VW Automotive Security Conf. Ingolstadt, Germany (2009)

[8] Hoppe, T., Kiltz, S., Dittmann, J.: Adaptive dynamic reaction to automotive it security incidents using multimedia car environment. In: 4th Int. Conf. on Information Assurance and Security. pp. 295–298. IEEE (2008)

[9] Kargl, F., Papadimitratos, P., Buttyan, L., Muter, M., Schoch, E., Wiedersheim, B., Thong, T.V., Calandriello, G., Held, A., Kung, A., et al.: Secure vehicular communication systems: implementation, performance, and research challenges. Communications Magazine 46(11), 110–118 (2008)

[10] Kleberger, P., Olovsson, T., Jonsson, E.: Security aspects of the in-vehicle network in the connected car. In: Intelligent Vehicles Symposium (IV). pp. 528–533. IEEE, Baden Baden (2011)

[11] Koscher, K., Czeskis, A., Roesner, F., Patel, S., Kohno, T., Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H.: Experimental security analysis of a modern automobile. In: IEEE Symp. Security and Privacy. pp. 447–462. Oakland, CA (2010)

[12] Larson, U.E., Nilsson, D.K., Jonsson, E.: An approach to specification-based attack detection for in-vehicle networks. In: Intelligent Vehicles Symposium. pp. 220–225. IEEE, Eindhoven, Netherlands (2008)

[13] Lazarevic, A., Kumar, V., Srivastava, J.: Intrusion detection: A survey. In: Kumar, V., Srivastava, J., Lazarevic, A. (eds.) Managing Cyber Threats, Massive Computing, vol. 5, pp. 19–78. Springer US (2005), http://dx.doi.org/10.1007/0-387-24230-9_2

[14] Matsumoto, T., Hata, M., Tanabe, M., Yoshioka, K., Oishi, K.: A method of preventing unauthorized data transmission in controller area network. In: Vehicular Technology Conf. (VTC Spring). pp. 1–5. IEEE, Yokohama, Japan (2012)

[15] Miller, C., Valasek, C.: Adventures in automotive networks and control units. Last Accessed from h ttp://illmatics. com/car_hacking. pdf on (2013)

[16] Miller, C., Valasek, C.: A survey of remote automotive attack surfaces. Last Accessed from http://illmatics.com/remote attack surfaces.pdf (2014)

[17] Muter, M., Asaj, N.: Entropy-based anomaly detection for in-vehicle networks. In: Intelligent Vehicles Symposium (IV). pp. 1110–1115. IEEE, Baden Baden, Germany (2011)

[18] Muter, M., Groll, A., Freiling, F.C.: A structured approach to anomaly detection for in-vehicle networks. In: 6th Int. Conf. Information Assurance and Security (IAS). pp. 92–98. IEEE, Atlanta, GA (2010)

[19] Nilsson, D.K., Larson, U.E.: Simulated attacks on can buses: vehicle virus. In: Proc. 5th IASTED Int. Conf. on Communication Systems and Networks. pp. 66–72. Langkawi, Malaysia (2008)

[20] Smith, R., Estan, C., Jha, S.: Xfa: Faster signature matching with extended automata. In: IEEE Symposium on Security and Privacy. pp. 187–201. IEEE (2008)

[21] Studnia, I., Nicomette, V., Alata, E., Dewarte, Y., Kaâniche, M., Laarouchi, Y.: Survey on security threats and protection mechanisms in embedded automotive networks. In: 2nd Workshop on Open Resilient Human-aware Cyber-Physical Systems. Budapest, Hungary (2013)

[22] Wolf, M., Weimerskirch, A., Wollinger, T.: State of the art: Embedding security in vehicles. EURASIP Journal on Embedded Systems (1) (2007)

[23] Yu, S.: State complexity of regular languages. Journal of Automata, Languages and Combinatorics 6(2), 221–234 (2001)