Change your Car's Filters: Efficient Concurrent and Multi-Stage Firewall for OBD-II Network Traffic

Felix Klement
Chair of Computer Engineering
University of Passau, Germany
felix.klement@uni-passau.de

Henrich C. Pöhls
Chair of IT Security
University of Passau, Germany
hp@sec.uni-passau.de

Stefan Katzenbeisser
Chair of Computer Engineering
University of Passau, Germany
stefan.katzenbeisser@uni-passau.de

***Abstract*—Modern cars offer one common interface to the outside, the OBD. Among the multitude of protocols that could exchange messages with the car's internal devices over OBD the CAN-BUS protocol is the most well-known; several commercial devices (so-called dongles) would allow to send and receive messages without any user-controlled restrictions. In order to enable fine-grained filtering on the CAN-BUS we exploit a security weakness called man-in-the-middle: the car or dongle does not apply any origin authentication as neither digital signatures nor message authentication codes (MACs) are used. We are the first to present this approach and offer measurements for our concurrent and multi-stage design that enables a fine-grained and extensible filtering approach for all protocols within the OBD.**

***Index Terms*—Network Security, Information Flow Control, Vehicular Security, CAN, OBD-II**

I. Introduction

The on-board diagnostic system (OBD) is a vehicle diagnostic system and the de facto interface for communication with the external environment. OBD supervises all exhaust gas influencing systems during operation. Furthermore, other important electronic control units (ECUs) are monitored. This allows for example reading the current speed, rpm and other information from the car. Occurring errors are communicated to the driver via different methods (e.g. control lights, audio signal etc.) and are persistently stored in the respective control unit. Thus, error messages can be retrieved via standardized interfaces. While most OBD-II communication today happens over ISO 15765 (CAN-BUS), a total of five protocols exist and can be used over OBD-II: ISO 15765-4 (CAN-BUS) [1], ISO 14230-4 (KWP2000) [2], ISO 9141-2 [3], SAEJ 1850 (VPW) [4], SAEJ 1850 (PWM) [4].

With all the above-mentioned functions car drivers are tempted to connect dongles to this port and let those dongles retrieve/send data via the OBD interface. The first attack that comes to mind would be a malicious dongle that sends false commands to the car, but even if information flows only in the other direction this allows attacks especially against the privacy of drivers[1]. Note, the data that can be retrieved from the car is a treasure trove, it includes drivers' habits that can be considered personal information. Thus drivers should be able to control and limit the data flow.

In this paper, we exploit the fact that the OBD interface lacks basic data authentication mechanisms, making it possible to place a firewall as a man-in-the-middle to allow fine granular filtering of the respective protocols. Effects of such filtering on commercially available dongles can be shown, such that they can not trust the car's data and often use secondary sources, like sensors from an accompanying smartphone application [5].

The de facto standard used for communication today is, as already mentioned, the CAN-BUS. Therefore, we will also use this technology for the first prototype of such a generic filter and for testing its performance. More precisely, we have a working software implementation as well as suitable hardware for ISO 15765. Nevertheless, our approach as well as the system underlying it is designed to be easily extensible to all protocols mentioned above as well as any individually modified implementations thereof. You need a hardware interface (see Section IV) and the appropriate software counterpart, the so-called protocol bindings, to support any kind of protocol.

II. Related Work

There are several works ([6], [7] and [8]); showing the wealth of information that can be extracted from a vehicle. They demonstrate how to monitor cars, predict the state of internal hardware, detect driving behaviour and detect various anomalies. The predominant architecture as well as the threats that dominate in modern vehicles was investigated by Wolf et al. [9]. They found that the gateways built into the automotive network require the use of powerful firewalls. Furthermore, they found that the firewall implemented in the gateways must also have rules that control access based on the security relevance of the particular network. Something like filtering

The authors acknowledge the financial support by the Federal Ministry of Education and Research of Germany in the program of "Souverän. Digital. Vernetzt." Joint project 6G-RIC, project identification number: 16KISK034. Additionally, this work was partially funded by the Bavarian Ministry of Science and Arts under the ForDaySec project.

[1]See: Connected Vehicles – Are Commuters in The Privacy Driving Seat? by S. Meyer; Nov.2018; https://www.cpomagazine.com/data-privacy/connected-vehicles-are-commuters-in-the-privacy-driving-seat/ [Sept. 2022]

approaches or firewall concepts for in-vehicle networks do exist. Still, the amount of available research is very limited, especially compared to work on cross-vehicle networks like VANETS (vehicular area networks). Unfortunately, there is even less information about existing solutions for vehicles. As an example, the manufacturer NXP describes the need to protect the car's networked devices from unwanted outside traffic by a gateway for "filtering inbound and outbound network traffic based on rules, disallowing data transfers from unauthorized sources."[10]. NXP further states that a more fine-granular approach "[...] may include context-aware filtering" [10]. However, how the exact mechanisms as well as the security functions work or are used in real vehicles is often not publicly disclosed. Manufacturer solutions such as the "Central Gateway" for central communication in the vehicle from Bosch, which lists a firewall and an intrusion detection system on its product page [11], unfortunately, do not disclose any exact details on the respective product pages. Even when specifically asked by the responsible department, we were unable to obtain any further information on the security features mentioned. An approach that acts as a gateway between the individual access points of a car was presented by the company Karamba Security [12] in 2016. Here, the developers integrate a system directly into the firmware. This is intended to prevent malicious code from being infiltrated into the system. Each control unit sets its own guidelines and generates a so-called whitelist with permitted program binaries, processes, scripts and network connections. Rizvi et al. [13] presents in the literature a distributed approach for a firewall system that can be used in automotive networks. Here, the approach focuses on ensuring that only valid packets reach an internal device by using a hybrid security system (HSS).

There are now some handy OBD-II devices on the consumer market that allow inbound filtering or simply blocking of all access. The main feature touted by these devices as being blockable is the use of "key duplicators and an accessible OBD-II socket", which would allow car thieves to generate new access codes, rendering the original keys obsolete. This creates a security barrier between the external devices and the data bus, protecting vehicle functions from unauthorized access and tampering. In virtually all commercially available products ([14], [15], [16], [17], [18], [19], [20]), there are almost always only two modes: either always deny (Off) or always allow (On). Their focus is generally on preventing malicious senders' packets from reaching important devices in the vehicle by disabling access to the CAN bus. Of course, this would also completely block data that could go the other way, but this privacy impact is not disclosed. In addition, existing approaches do not allow the use of an OBD-II dongle in an activated state, as no data is available for processing. Our approach closes this gap.

III. Architecture

The architecture is generally divided into two parts. One consists of all the software components that are responsible for providing protocol-specific communication as well as the actual execution of traffic filtering. The other part consists of the associated hardware to address the respective protocol interface from the hardware side.

A. Software Components

A complete detailed specification of all software components would be inappropriate and out of scope. Nevertheless, we would like to briefly explain the basic ideas of the respective components in theory and show why we choose exactly this approach. Therefore, we will briefly outline the most important parts in the next sections.

1) Producer/Consumer Scheme: The producer-consumer problem (also known as the bounded buffer problem) is a classic example of a multi-process synchronization issue, the first version of which dates back to Edsger W. Dijkstra in 1965. Nevertheless, there are now promising approaches in software development to efficiently eliminate this problem [21]. We have decided on such an approach (more details can be found in Section VII-A). A filtering approach is best realized with a buffer in which incoming messages are accumulated. Afterwards, they can be processed one after the other, depending on the queue. To be as unrestricted as possible in terms of processing, this model is also perfect. Depending on the respective computing power, several producers or consumers can be started. In this way, load peaks can be easily absorbed. The modular approach can also be applied by means of differently implemented producers.

2) Modular Approach for Protocol Bindings: Since the producer/consumer scheme allows us to easily create several differently implemented producers, a uniform interface must be defined. This interface ensures that the responsible consumer can correctly process and forward incoming messages. By means of this approach, it is possible to support incoming messages of all protocols.

3) CAN-Bus Binding: In order to support the CAN protocol for our implementation, a connector is needed to receive messages as well as to be able to send filtered or processed messages again. For this purpose, already used libraries as well as the common syntax for encoding and decoding are used. Furthermore, our binding is going to understand the so-called DBC format. DBC stands for CAN Database and is a proprietary format that describes the CAN bus data structure.

4) Processing Pipeline: A concurrent and multi-level data input and data processing pipeline is used for the processing pipeline. This makes it possible to efficiently consume different sources, the so-called producers. Here, it is also possible to configure the processing pipeline with regard to the resources to be used. More precisely, the number of processes for producer and consumer as well as the concurrency and the batch size to be used. In Figure 1 you can see a schematic overview of the pipeline. As already mentioned, it will be possible to develop a producer for each protocol. So in the future, there may be a producer for CAN, one for ISO9141 and so on. There is a uniform interface to adhere to. The producers then send their messages to the concurrent and multi-stage data ingestion service. There, the incoming messages are

analyzed and then asynchronously serialized and inspected. Here, serialization refers to the application of the active rules and not to the quantity of messages.

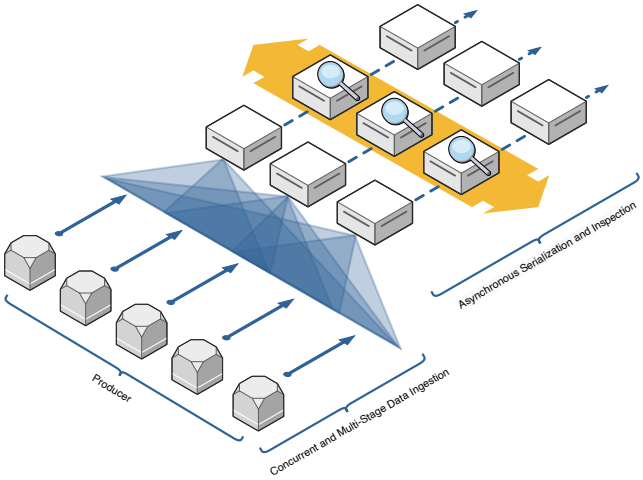

Fig. 1. Simplified representation of the processing pipeline

5) Serialization: After the incoming messages have been serialized and bundled into batches, the messages are checked for the active rules as efficiently as possible. Since the behaviors of a rule can be sorted according to their strictness in the restriction, the strictest behaviors will apply. This allows the individual behaviors to carry out the checks in parallel.

IV. Interfacing the OBD-II Firewall

One of the most important parts of our firewall is the connection to the OBD-II interface. The connection from our firewall to the outside should not have any difference for the end-user compared to the conventional OBD-II connector. Our final prototype is a setup with a female connector port and a male adapter cable. The female port will be used for connecting devices that are filtered by our firewall. The male connector will be used to connect the firewall to the existing OBD-II interface of the respective vehicle. In order to implement our filtering approach safely, we decided on a clear physical separation between the CAN interface for the incoming messages (i.e. the interface where you then connect the dongle) and the CAN interface which is responsible for sending or passing on the data to the OBD-II interface. This separation concept makes it possible to have full control over all incoming and outgoing traffic (see Figure 2).

V. Policy Management

Policies are managed using configurations specified in JSON format. We do not currently use any particular rule framework or rule engine. The format is a simple one that is adapted to the current use case but can be extended in a modular way. The current overall structure can be seen in Tables I and II.

To support future extensions a kind of version check is implemented to check the respective rules before being applied. Table I shows the overall wrapper structure for a rule definition. This contains general information such as a description, the protocol type to be filtered and the version of the policy language currently in use. Table II describes the structure of a so-called behavior. A rule can theoretically have as many behaviors capsules as desired.

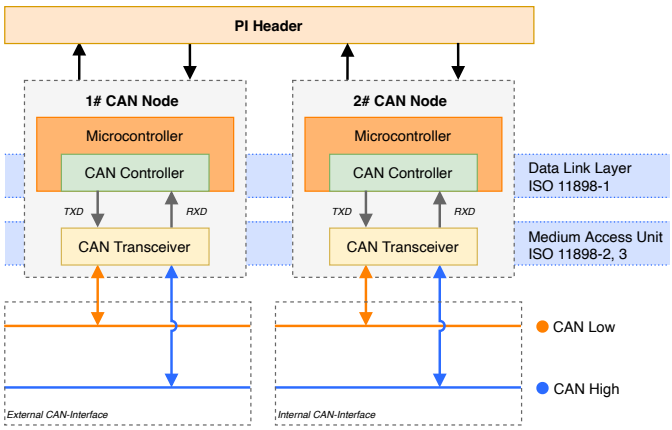

Fig. 2. Separation concept for the CAN bus within our firewall

TABLE I
List of basic properties with their associated functionality

Property	Type	Description
name	*<String>*	Is just a simple naming of the individual rules for better distinction. The name does not have to be unique.
description	*<String>*	Briefly describes the created rule in a few words.
version	*<String>*	The version number specifies the version of the properties to be used.
protocol	*<Protocol-Type>*	Declares the protocol type to be used for the respective rule. Currently there is only *<CAN>* as a declarable type. However, due to the modular approach, more types may be added in the future.
behaviours	[*<Behaviour>*]	The behaviour field defines a list of all actions to be performed later during the execution of each rule. More information about the types available within a behavior can be found in Table II.

TABLE II
List of properties for a single behavior inside a policy

Property	Type	Description
type	*<String>*	Currently, three different behaviour types are supported: • *reject* - Ignores all messages with the defined identifier and associated value • *limit* - Limits all accruing values of a message from the defined identifier by means of a predefined value • *replace* - Always exchanges all message values of a given identifier with the given value
identifier	*<String>*	Defines the identifier of the CAN message present on the bus
value	*<String>*	Determines the data payload to be used for the respective set type
The following properties are optional and do not have to be set		
delay	*<Integer>*	If the delay property is set, all messages that fall below the specified behavior will be delayed. The value is given as an integer value and defines the delay time in milliseconds
pub_once	*<Boolean>*	Allows messages in the scope of the behavior to be allowed only once per system start. Once the message has been read once, it is whitelisted and then not forwarded. By default, the value is set to *false*.
id_range	*<String>*	By means of the identifier range, the behavior value range to be enforced can be extended.
val_range	*<String>*	Allows messages in the scope of the behavior to be allowed only once per system start. Once the message has been read once, it is whitelisted and then not forwarded. By default, the value is set to *false*.

VI. Rule enforcement

The enforcement of the rules as well as the individual behaviors is based on an assessment of importance. This means that more important behaviors and rules outweigh lower ones. Furthermore, it is possible to easily deactivate and activate individual rules. Also, it is interesting to get some specific metrics related to rule/behaviour enforcement. Therefore, the number of filtered, modified and blocked messages is tracked.

VII. Implementation

A. Producer/Consumer Solution

As already described in Section III-A1, our approach follows the so-called producer-consumer construct and thus makes a modular approach possible. This is one of the disciplines in which Elixir can demonstrate all its abilities and advantages in the best possible way.

We are using a concurrent, multi-stage for building data input and data processing pipelines. This enables developers during testing to efficiently use data from various sources, such as Amazon SQS, Apache Kafka, Google Cloud PubSub, RabbitMQ and others. It is also possible to implement your own producers. Depending on the scenario, you may want to process messages as batches (groups of messages) before publishing the data. This is also beneficial in our use case: While we don't need the batchers to communicate with an extraneous API, it allows us to process the storing of CAN messages in an encapsulated way and thus have no runtime loss for publishing already filtered CAN messages. This allows for increased throughput and consequently improved overall performance of our pipeline. Batches are simply defined via the configuration option.

Our configuration declares the following pipeline:

- 1 CAN producer
- 2 CAN processors *(to show the concurrency approach)*
- 1 batcher `:batcher_can_bus` with 1 batch processor
- 1 batcher `:batcher_db_storage` with 1 batch processors

This configuration is of course adaptive and can be extended very easily if required. A schematic representation of how the pipeline looks exactly is shown in Figure 3. There, the simple structure and how the individual pipeline components are interdependent become clear.

At the end of the pipeline, the messages are automatically acknowledged. If no batchers are configured, the acknowledgment is carried out by processors. The number of messages acknowledged, assuming the pipeline is running at full capacity, is `max_demand - min_demand`. Since the default values are 10 and 5 respectively, we will be acknowledging in groups of 5. Since there are batchers in our case, the confirmation is done by the batchers using the configured `batch_size`. In case of errors, the failed message or batch is instantly acknowledged as failed. A log report is also issued for each error. In addition, we have defined the `handle_failed/2` callback in our module. This callback is called by all failed messages before they are acknowledged. This allows including additional error handling mechanisms if needed in the future or implementing correct error behavior for other protocols.

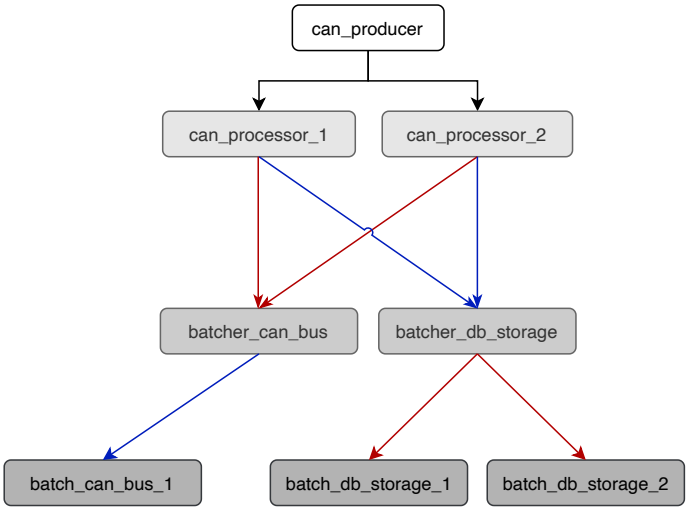

Fig. 3. Representation of producer, processor and batcher pipeline

The interplay between producer and consumer and the components responsible for reading and writing CAN messages can be seen in Figure 4. The `:InputStreamReader` accepts all messages arriving on the respective defined CAN interface and processes them in our library. The module then holds a stream ready from which accumulated messages can be processed at any time. The `:Consumer` creates a message handler after its parent process has been started, which then requests a specific message demand D_n from the `:Producer`. The `:Producer` then starts to transform D_n messages from the `:InputStreamReader` stream into message structs, which are then processed by the `:ConsumerMessageHandler`. The actual filtering of the individual CAN messages then takes place within this handler (more on this in Section VII-C). Subsequently, the filtered messages are transferred to the message batcher of the `:Consumer` to be further processed there, if necessary. As can be seen in Figure 3, there is the `batch_can_bus_1`. This then writes the transferred messages back to the defined CAN bus using the `:CAN-BusWriter`.

B. Rule Parsing

To check the correctness of the created rules before actually applying them, we have created a rule parser. The general parser structure is again designed in such a way that it is possible to create further parsers for future protocols on the basis of a uniform interface in order to be able to adapt to possible changes in the respective rules or behaviors for other protocols. Each parser must have a `parse/1` function defined that takes the current rule configuration as input. Within this function, the structure for the respective protocol must then be defined schematically. Listing 1 shows our defined scheme for the CAN rule and its individual behaviors.

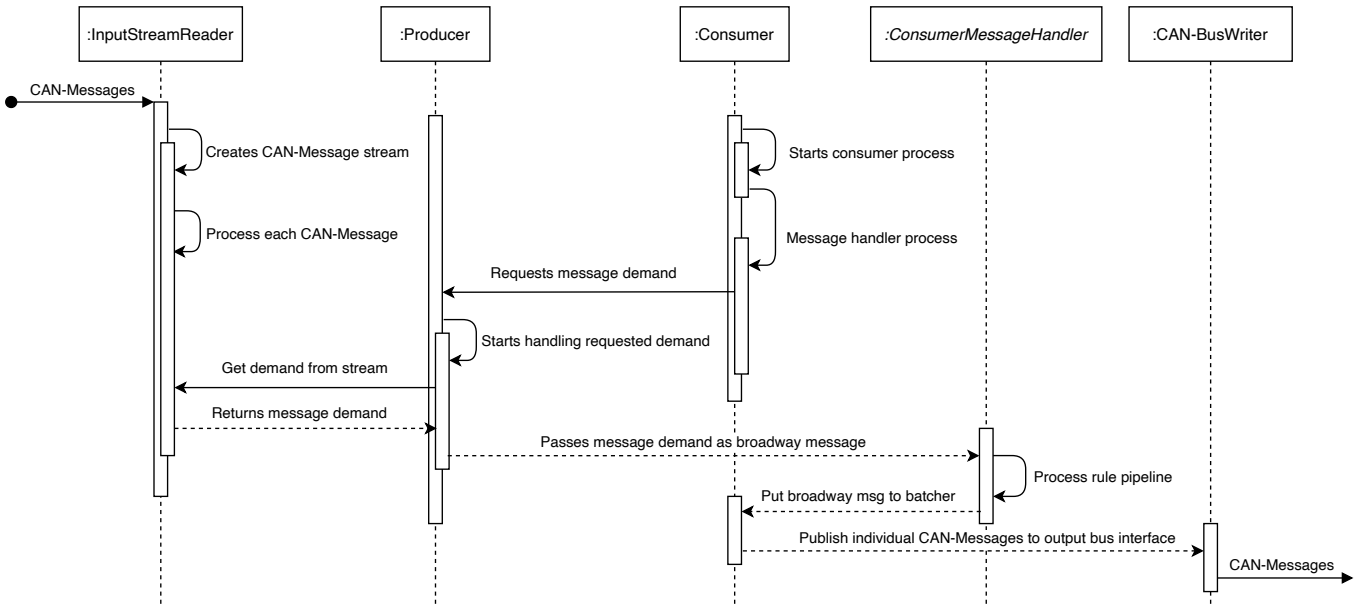

Fig. 4. Sequence diagram of basic producer/consumer procedure

```
1  rule_schema = %{
2   "name" => :string,
3   "description" => :string,
4   "protocol" => [:string, Validators.validate_protocol("
        CAN")],
5   "behaviours" => [
6    :list,
7    :map,
8    :required,
9    %{
10   "type" => [:string, Validators.validate_can_type()],
11   "identifier" => [:string, Validators.
         validate_can_identifier()],
12   "value" => [:string, Validators.validate_can_value()
         ],
13   "delay" => [:string, Validators.validate_can_value()]
14   }
15  ]
16 }
```

Listing 1. Example of what a rule scheme looks like based on our implementation for the CAN protocol

In order to carry out our more specific checks, we create so-called validators for individual fields. Basically, all desired checks can take place within these validators. In Listing 2 you can see an example of our definition of a validator for a protocol. Here we simply pass the desired protocol name as `:String` and compare it with the value defined in the rule for `"protocol"`. This allows our approach to validate additional protocols as individually as necessary.

When starting the `CanConsumer`, the configuration is initially passed to the `CanParser`. The parser then checks whether the configuration is valid by means of the defined schema and the respective validators which are loaded via the corresponding `Validators` module. A simplified overview of the relationships between the modules can be found in Figure 5. If the configuration of the rules is valid, it is stored in an agent process. We implemented a basic server with which the state can be retrieved and updated via a simple API. This allows us to easily change the configuration during runtime without having to stop or restart the firewall. Using the API, the current configuration is then retrieved in the filtering pipeline.

```
1  def validate_protocol(protocol)
2      when is_binary(protocol) do
3   fn data ->
4    bool = data == protocol
5    if bool do
6     :ok
7    else
8     {:error, "Expected => #{protocol} / Got => #{data}"}
9    end
10  end
11 end
```

Listing 2. Custom validator for testing if the right protocol is defined in the rule schema

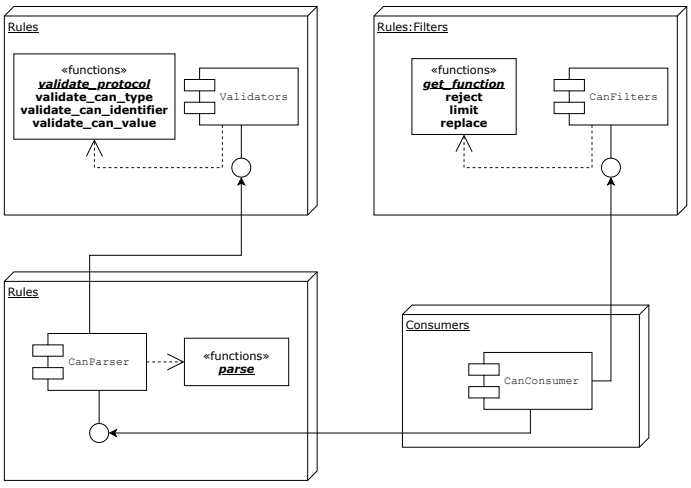

Fig. 5. Structure and interaction of rules module with consumer module

C. Filtering Approach

Our filter approach consists of a single pipeline. We express computations on collections, although the computations are performed using multiple stages in parallel. It is designed to work with both limited (finite) and unlimited (infinite) data. By default, our approach works with batches of 500 items. We also offer the concepts of "windows" and "triggers". This allows data to be split into arbitrary windows according to event time. With triggers, calculations can be materialized at different intervals, so it is possible to have a look at the results while they are being calculated. Since our producer provides a theoretically unbounded stream of CAN messages, this is a perfect fit to allow efficient computation or filtering of individual messages.

Roughly summarised, the pipeline can be broken down into three main stages. The first step is to load the current `behaviors` from the agent and check whether the current message identifier occurs in the `behavior` identifiers. If this is the case, the applicable `behaviors` are sorted according to their type in step two. Each type has a rating R_B which is used for sorting, whereby 0 has the highest priority and subsequently is sorted in descending order. For our CAN implementation, the following ratings are used to determine the order of preference:

- `:reject` - $R_B = 0$
- `:replace` - $R_B = 1$
- `:limit` - $R_B = 2$

In the same step, a map is created using the rating R_B as the key. The associated value also consists of a map with the keys `:function` for the associated filter function and `:payload` for the defined `behaviour` value. Then, in step 3, the highest priority filter function defined by R_B is selected and applied to the CAN message. The last step in the pipeline is to pass the filtered message to the batchers. There, the `Compostor` of our library will write the filtered message to the desired CAN interface.

VIII. Benchmarks

In terms of benchmarking and performance testing, we have limited ourselves to the performance of the actual filtering pipeline. All our tests were performed on a Raspberry Pi with a 64-bit Quad Core ARM v8 Cortex-A72 @ 1,5 GHz. We execute each of the desired functions for a certain time after an initial warm-up phase and then measure their runtime and optionally the memory consumption.

In Table III you can see the measurement results for the runs of our pipeline with different numbers of `behaviors`. The number of measurements is always a multiple of three, as we currently have three different types for a single `behavior`. Therefore, in order to ensure that we always have an equal number of types of each type, this multiple of three is the result. The measurements generally show that our filtering pipeline only requires a small amount of computing time and therefore has no impact on usability. The iterations per second measure how many times we could run the full pipeline within one second. Figure 6 visualizes this and shows expected linearity: the average runtime of the pipeline increases with an increase in the number of `behaviors` (which increases the number of rules to be parsed) defined within a rule.

Fig. 6. Run time boxplot for each measurement

The maximum number of `behaviors` used in our measurements is 30000. This is the chosen upper limit for the measurements. While being large compared to the foreseeable application, the average execution time for 30000 `behaviors` within a single rule is `59.72 ms`. This is fast enough for the intrinsic passive rule of a dongle. In itself, a slight delay in the extreme case of 0.06 seconds is more than acceptable, because during our functional tests using the RYD-Box [22] and the Volkswagen (VW) Data Plug [23], we noticed that often the interfaces that pass data to the outside of the vehicle or to third parties are not time-critical in nature. Most of the time, it is only a matter of connecting peripheral diagnostic devices, e.g. via OBD-II, providing information to the infotainment system, or the provision of data for cloud services such as monitoring the car via a mobile phone app. Thus, the measured delays due to our filtering of data can be neglected for the applications of the commercial dongles we tested.

IX. Conclusion

Looking at related work, this is the first approach to enable fine-grained ingress and egress filtering of messages flowing over the vehicle's OBD interface. We build and tested an approach that facilitates a man-in-the-middle security weakness to block or even selectively modify CAN-BUS messages based on rules. Using the Elixir programming language and a Raspberry Pi we build a concurrent and multi-stage OBD-II data ingestion filtering approach with asynchronous serialization that can be flexibly extended to cover different OBD protocols in the future. While there have been approaches for a firewall in a car, none of the existing works offer fine-grained solutions. We provide measurements for our prototype. Our measurements show a tolerable delay of about 5 ms for the most likely number of `behaviors` (i.e. one `behavior` is one action in a rule) in a rule using non-optimized hardware such as a Raspberry Pi. Tolerable because the current commercial dongle's use cases are rather to report or log driving behaviour for various purposes.

TABLE III
RESULTS OF THE RUN TIME COMPARISON OF THE SINGLE MEASUREMENTS

Measurement	Iterations per Second	Average	Deviation	Median	Minimum	Maximum	Sample Size
3 `behaviours`	243.87	4.10 ms	±9.71%	4.08 ms	3.05 ms	6.79 ms	1218
30 `behaviours`	225.89	4.43 ms	±11.89%	4.36 ms	3.29 ms	10.23 ms	1218
300 `behaviours`	164.58	6.08 ms	±11.90%	6.03 ms	4.45 ms	11.07 ms	823
3000 `behaviours`	83.22	12.02 ms	±7.44%	12.02 ms	9.55 ms	14.68 ms	416
30000 `behaviours`	16.74	59.72 ms	±7.09%	59.10 ms	51.62 ms	69.53 ms	84

REFERENCES

[1] *Road vehicles — Diagnostic communication over Controller Area Network (DoCAN) — Part 4: Requirements for emissions-related systems*. Standard IS0 15765-4. Geneva, CH: International Organization for Standardization, Feb. 2011.

[2] *Road vehicles — Diagnostic systems — Keyword Protocol 2000 — Part 4: Requirements for emission-related systems*. Standard ISO 14230-4. Geneva, CH: International Organization for Standardization, June 2000.

[3] *Road vehicles — Diagnostic systems — Part 2: CARB requirements for interchange of digital information*. Standard ISO 9141-2. Geneva, CH: International Organization for Standardization, Feb. 1994.

[4] *Class B Data Communications Network Interface*. Standard SAE J1850. Pennsylvania, USA: International Organization for Standardization, Oct. 2015.

[5] F. Klement, H. C. Pöhls, and S. Katzenbeisser. "Man-in-the-OBD: A modular, protocol agnostic firewall for automotive dongles to enhance privacy and security". In: *5th International Workshop on Attacks and Defenses for Internet-of-Things (ADIoT 2022) at ESORICS*. Springer, 2022.

[6] A. El Basiouni El Masri, H. Artail, and H. Akkary. "Toward self-policing: Detecting drunk driving behaviors through sampling CAN bus data". In: *2017 International Conference on Electrical and Computing Technologies and Applications (ICECTA)*. 2017, pp. 1–5.

[7] B. Nirmali et al. "Vehicular data acquisition and analytics system for real-time driver behavior monitoring and anomaly detection". In: *2017 IEEE International Conference on Industrial and Information Systems (ICIIS)*. 2017, pp. 1–6.

[8] A. Srinivasan. "IoT Cloud Based Real Time Automobile Monitoring System". In: *2018 3rd IEEE International Conference on Intelligent Transportation Engineering*. 2018, pp. 231–235.

[9] M. Wolf, A. Weimerskirch, and C. Paar. "Security in Automotive Bus Systems". In: 2004.

[10] *Automotive Gateway: A Key Component to Securing the Connected Car*. NXP Semiconductors, Feb. 2018.

[11] Bosch. *Bosch Central Gateway*. 2022. URL: www.bosch-mobility-solutions.com/en/products-and-services/passenger-cars-and-light-commercial-vehicles/connectivity-solutions/central-gateway/ (visited on 05/24/2022).

[12] Karamba Security. *Karamba Security - Homepage*. URL: https://karambasecurity.com (visited on 04/08/2021).

[13] Syed Rizvi et al. "Protecting an Automobile Network Using Distributed Firewall System". In: *Proceedings of the Second International Conference on Internet of Things, Data and Cloud Computing*. ICC '17. Cambridge, United Kingdom: Association for Computing Machinery, 2017. ISBN: 9781450347747. DOI: 10.1145/3018896.3056791.

[14] The Diagnostic Box. *OBD blocker*. 2022. URL: https://thediagnosticbox.com/product.php?pc=%5C%5COBD+Blocker (visited on 07/26/2022).

[15] Ampire. *CAN-Firewall*. 2022. URL: https://www.ampire.de/-WFS300-BT.htm?SessionId=%5C&a=article%5C&ProdNr=%5C%5FWFS300-BT%5C&p=1857 (visited on 07/26/2022).

[16] Ampire. *OBD-Firewall*. 2022. URL: https://www.ampire.de/Product-Archive/Ampire/Theft-protection/AMPIRE-OBD-Firewall-without-harness-.htm?shop=ampire%5C_en%5C&SessionId=%5C&a=article%5C&ProdNr=%5C%5FOBD-FIREWALL%5C&p=1857 (visited on 07/26/2022).

[17] Paser. *Firewall OBD2*. 2022. URL: https://automotive.paser.it/en-gb/Paser/Firewall-OBD2-card-included-p1069m11.html (visited on 07/26/2022).

[18] UniversClug. *Electronic Anti-thefts systems*. 2022. URL: https://www.universclub.com/en/buy/cat-electronic-anti-thefts-systems-2164.html (visited on 07/26/2022).

[19] AutoCYB. *Vehicle cybersecurity lock*. 2022. URL: https://autocyb.com/product/%5C%5Cautocyb-vehicle-cybersecurity-lock/ (visited on 07/26/2022).

[20] CAN Hacker. *Automotive diagnostic firewall*. 2022. URL: https://canhacker.com/projects/obd2-diagnostic-firewall/ (visited on 07/26/2022).

[21] L. Lamport. "A New Solution of Dijkstra's Concurrent Programming Problem". In: *Concurrency: The Works of Leslie Lamport*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 171–178. ISBN: 9781450372701.

[22] ThinxNet GmbH. *RYD Box*. 2022. URL: https://de.ryd.one (visited on 07/26/2022).

[23] Volkswagen. *Volkswagen Data Plug from Texa*. 2022. URL: https://www.volkswagen.de/de/konnektivitaet-und-mobilitaetsdienste/konnektivitaet/we-connect-go.html (visited on 07/26/2022).