# PAVFuzz: State-Sensitive Fuzz Testing of Protocols in Autonomous Vehicles

Feilong Zuo[†], Zhengxiong Luo[†]*, Junze Yu[‡], Zhe Liu[§], Yu Jiang[†]
[†]KLISS, BNRist, School of Software, Tsinghua University,
[‡]Beijing University of Posts and Telecommunications, [§]Nanjing University of Aeronautics and Astronautics

***Abstract*—The rapid development of in-vehicle networks and protocols brings efficient communication service but also increases the risk of attack. Any vulnerability may be leveraged to cause serious consequences. It is of vital importance to guarantee their security. However, the vulnerability detection efficiency of traditional techniques such as fuzzing is challenged by the complex relations among protocol states.**

**In this paper, we propose `PAVFuzz`, a state-sensitive fuzz testing framework to secure those protocols used in autonomous vehicles. It automatically learns relations between two data elements in different protocol states. The relations will then be used to calculate and update the mutation weight of each data element continuously. Accordingly, `PAVFuzz` is able to select the target data elements and perform state-sensitive mutation to boost the efficiency. Experiments show that, compared with state-of-the-art fuzzers Peach and AFL, `PAVFuzz` increases branch coverage by averagely 22.51% and 369.19% within 24 hours. It has successfully exposed 12 serious previously unknown vulnerabilities among several protocols that are widely used in autonomous vehicles, such as RTPS and SOME/IP. We have reported them to the developers and corresponding patches have been released.**

***Index Terms*—State-sensitive Fuzzing, Protocol Testing, Vulnerability Detection, Autonomous Vehicle**

## I. Introduction

The rapid development of autonomous vehicles puts forward higher communication requirements and various protocols have been developed to achieve efficient communications among different system components. However, these protocols also increase the risk of attack. Attackers may leverage the vulnerabilities of the protocol to achieve malicious purposes, such as deactivating ECUs and stealing the control right. For example, the vulnerabilities in the Jeep Uconnect system once gave the chance for attackers to control the whole vehicle [11]. It is of significant importance to guarantee the security of protocols in autonomous vehicles.

Recently, fuzz testing, as one of the most famous software testing techniques for vulnerability detection, has been included in the Society of Automotive Engineers J3061 standard [16] for cybersecurity guidance. Fuzzers usually generate large numbers of test inputs for execution, and the system under test is monitored for abnormal behaviors. According to the ways how new test inputs are produced, fuzzers can be divided into two categories: mutation-based and generation-based. Mutation-based fuzzers, such as AFL [24] and LibFuzzer [6], start with some initial seeds and constantly mutate them at byte/bit level to produce new inputs. Generation-based fuzzers, represented by Peach [17] and Sulley [1], require user-provided data models to stipulate the property of each specific data element and combine all elements together into complete test inputs. Both types of fuzzers have successfully detected a large number of vulnerabilities in real-world software, such as common libraries, utilities, browsers, etc.

However, when faced with the protocols in autonomous vehicles, the complex relations among protocol states bring two challenges:

**How to capture relations across different states.** Traditional fuzzers are usually state-insensitive, and treat each iteration and state equally and independently. But different from the traditional point-to-point network protocols, protocols in autonomous vehicles need to handle communication among various ECUs and sensors, therefore, they tend to employ a de-centered structure with multiple states of service discovery, service subscription, service request, and so on. These states are in close-relationship, where packet in one state not only defines the inner status of the protocol but also affects how the packets of the following states will be handled. Though provided with user-defined data models to describe the structure of packets of each single state, traditional fuzzers cannot deal with the relationship across different states, which can be leveraged to achieve a more accurate fuzzing procedure.

**How to perform targeted mutations in different states.** Traditional fuzzers treat each data element or byte/bit equally with the same probability to be mutated at each fuzzing iteration. However, considering the complex relations across different protocol states, the protocol state is influenced by previously sent packets. The influenced data elements of the state should be given a higher mutation opportunity. Therefore, to achieve a higher efficiency, the mutation strategy over data elements should be state-sensitive, not state-independent like in traditional fuzzers.

In this paper, we proposed a state-sensitive fuzzing framework named `PAVFuzz` to secure protocols used in autonomous vehicles. It mainly includes two components. First, the relation learning module automatically learns the relationship between two data elements in models of different states. The main idea is that, when a generated packet covers a new branch, it identifies the data elements contributing to this coverage increment and updates their relationship with the data elements of the following state. Second, the state-sensitive mutation module will perform targeted mutation on data elements. It no longer treats each data element with equal mutation weight, but calculates the mutation weight of each data element based on the continuously updated relation table. In this way, `PAVFuzz` is able to smartly recognize the target data elements that may trigger new branches, thus improving the efficiency of fuzzing.

We evaluated `PAVFuzz` on several popular open-source protocols used in autonomous vehicles: RTPS, SOME/IP, and ZeroMQ. Experimental results show that, compared with state-of-the-art fuzzers Peach and AFL, `PAVFuzz` improves the branch coverage averagely by 22.51% and 369.19%, respectively. Moreover, `PAVFuzz` exposed 12 previously unknown vulnerabilities that may cause serious consequences among these widely used in-vehicle protocols.

## II. Background

### A. Protocols in Autonomous Vehicles

In autonomous vehicles, various sensors and ECUs work together to collect environmental information and conduct control over vehicle components. Fundamental vehicular network systems connect them

*Zhengxiong Luo is the correspondence author.

978-1-6654-3274-0/21/$31.00 ©2021 IEEE

together and serve as the carrier for information exchange. The traditional and most widely-used vehicular network system is Controller Area Network (CAN). With the information getting richer and the requirement of bus speed getting higher, other systems came into existence, such as Local Interconnect Network (LIN), FlexRay, Media Oriented System Transport (MOST).

In recent years, 100BASE-T1 automotive Ethernet has been gradually used in autonomous vehicles. The automotive Ethernet uses two twisted-pair wires and provides the ability of information transmission up to 100Mb/s. Figure 1 shows the protocol stack of automotive Ethernet. Based on the basic Ethernet, many kinds of protocols are supported, like common TCP/IP on layer 3-4, Scalable service-Oriented Middleware on IP (SOME/IP) on layer 5-7, etc. Moreover, modern communication middleware like Real-time Publish-Subscribe Protocol (RTPS) and ZeroMQ are adopted to achieve high-throughput and low-latency communication in not only intra-vehicle but also inter-vehicle network systems, which also provide the possibility to be attacked maliciously if any security vulnerability exists in their implementations.

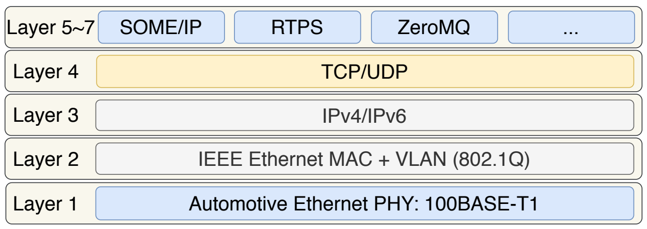

Fig. 1. Example Protocol Stack of Automotive Ethernet.

## B. Generation-based Fuzzing

For protocols, generation-based fuzzers are more suitable than mutation-based ones, because network packets are highly structured and the mutation-based ones often bog down without the help of data models. Specifically, the byte/bit-level mutation of mutation-based fuzzers tends to destroy the structure of packets and thus they will be rejected by the protocol at the early processing stage. Generation-based fuzzers such as Peach have achieved great popularity in the testing of many famous protocols such as FTP and Modbus, and their main workflow of packets generation is shown in Algorithm 1.

**Algorithm 1:** Packets Generation of Generation-based Fuzzer

```
Input: $Set_D$: Set of data models in order of states
Output: $Seq_p$: Sequence of packets generated in this iteration
1 $Seq_p \leftarrow$ EmptySequence()
2 for $\mathcal{D}_i \in Set_D$ do
3     $Set_{elem} \leftarrow$ RandomChooseElements($\mathcal{D}_i$)
4     for $elem \in Set_{elem}$ do
5         $\mathcal{D}_i \leftarrow$ ConductMutation($\mathcal{D}_i, Set_{elem}$)
6     $packet \leftarrow$ GeneratePacket($\mathcal{D}_i$)
7     $Seq_p \leftarrow Seq_p \bigcup \{packet\}$
8 return $Seq_p$
```

The process starts with user-provided data models, each of which represents the structure of packet that is sent to the under-test endpoint (e.g., a server endpoint in C/S protocols or a subscriber in P/S ones) at a specific state. Fuzzers traverse these data models in order of states (lines 2-7). At each data model $\mathcal{D}_i$, several data elements in $\mathcal{D}_i$ are randomly selected to be mutated according to the priorities of elements and the mutation rules (lines 3-5). Afterwards, a new packet is generated and added to the sequence (lines 6-7).

## III. Motivation

Take the RTPS protocol [13], a commonly used communication protocol in the automatic driving system like Adaptive Autosar [3] and Baidu Apollo [2], as an example. To fuzz the implementation of RTPS, testers will model the structure of packets to simulate the real communication process. Several data models will be constructed, including the participant discovery packet model (SPDP), the endpoint discovery packet model (SEDP), the data publishing packet model (PUB), and so on. Based on the specification of these data models, fuzzers could continuously generate data packets and monitor the system behaviours. During the fuzzing procedure, we find that the efficiency of the traditional state-insensitive strategies are impeded by the relations among those states.

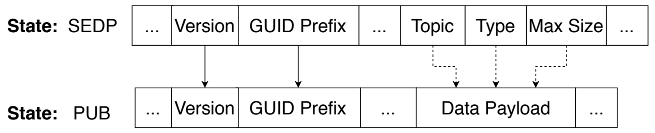

Fig. 2. Relations between data elements of different states in RTPS protocol.

Figure 2 presents some relations between data elements of two states in RTPS protocol. For example, the `Version` and `GUID Prefix` elements of PUB state should keep the same value as the previous packet in SEDP state. Otherwise, this packet tends to be rejected for being inconsistent with the system. In this situation, the mutation probability of these elements should be reduced to avoid useless mutation and execution. Another relation instance is more complicated, it is clear that the `Data Payload` element is the detailed content of the attributes `Topic`, `Type` and `Max Size`, which are stipulated in the previous packet in SEDP state. The change of values in the attribute elements will greatly influence the way how the protocol processes the content element. In this situation, the mutation probability of these elements should be increased to explore more possibilities. Besides these two relations, there are various other relations between states in RTPS. However, none of the traditional fuzzers capture those relations to improve fuzzing.

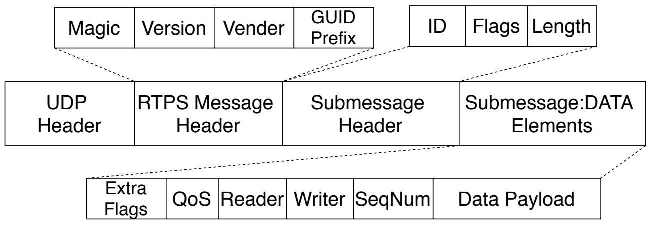

Fig. 3. Structure of RTPS packet used to publish data.

More specifically, Figure 3 shows the packet structure of PUB state that will be sent from an RTPS publisher to a subscriber. The structure is complex and this figure only describes the basic data elements. Indeed, some elements can be further subdivided, like the Flags, GUID Prefix, Data Payload, etc. Traditional generation-based fuzzers, like Peach, first generate initial seeds. Then, at each fuzzing iteration, randomly choose some data elements and mutate them to new values. At a superficial level, it seems reasonable as it does work in producing new test inputs. Nevertheless, considering the complex relations between states in RTPS, this mutation strategy is inefficient because it treats all elements in the data model equally. In fact, in the logic of packet analyzing, these data elements have different levels of significance. More specifically, the significance of each element is up to the former packets of SPDP and SEDP states.

## IV. PAVFuzz Design

We present the design of PAVFuzz in Figure 4. The whole system starts with a set of user-provided data models of the under-test protocol implementation. Each data model $\mathcal{D}$ represents the structure of the packet which is sent to the endpoint under-test (EUT, usually a server in C/S protocols or a subscriber in P/S protocols).

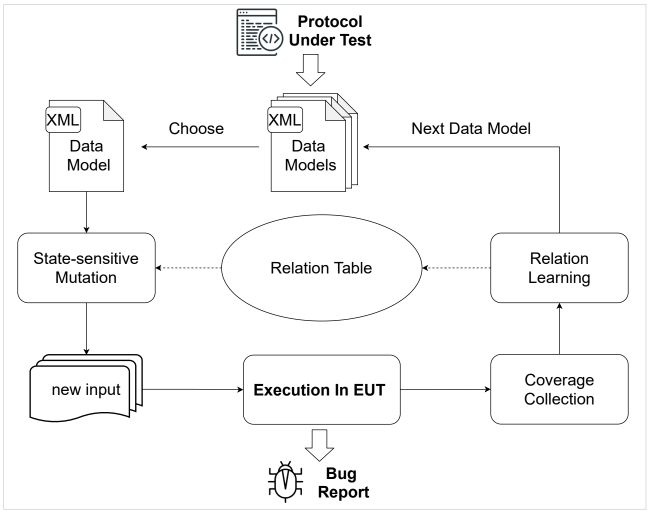

Fig. 4. Design of PAVFuzz. A relation table is dynamically maintained. When a generated packet covers new code, the relation learning module updates the relation table. The state-sensitive mutation module leverages the dynamic weight to preform target mutation over data elements.

At each iteration of the fuzzing loop, PAVFuzz traverses the data models in the order of states. When a specific data model $\mathcal{D}$ is chosen, it conducts state-sensitive mutation on target data elements according to the dynamic mutation weight calculated on the relation table to generate new input. Afterwards, the newly generated input is fed to the EUT and PAVFuzz monitors the status of the input processing. If any crashes are detected, it implies the possible existence of vulnerabilities in the code logic of the implementation, and PAVFuzz will report the related information of these crashes for reproducing and repairing. Otherwise, if no crashes arise, PAVFuzz moves to the coverage collection stage and decide whether new code in the protocol are covered during the packet processing just now. If new codes are covered, PAVFuzz leverages a light-weight strategy to learn the relations between the elements mutated in $\mathcal{D}$ and the elements in next data model $\mathcal{D}'$. The learned relations are quantified in the form of value in the relation table, whose content grows as the fuzzing goes on. After that, $\mathcal{D}'$ turns into the chosen data model to be mutated. When all data models have been traversed, PAVFuzz moves to the next iteration.

### A. Relation Table

The relation table in PAVFuzz describes the relations between data elements in data models of adjacent states. As Figure 5 shows, each cell in the table represents a structure of triple `<Element_ID_P, Element_ID, Value>`, where `Element_ID_P` uniquely identifies the element of the data model in the previous state and `Element_ID` has a similar meaning in the current data model. To achieve the uniqueness, a tuple structure of `<Model, Element>` is employed as the inner data structure of `Element_ID_P` and `Element_ID` in case of element conflict. `Value` in each cell dynamically reflects the relation between `Element_ID_P` and `Element_ID`. Taking the two globally unique element ids as $\mathcal{ID}_p$ and $\mathcal{ID}_c$, briefly speaking, `Value` means the amount of new code coverage that is achieved when combining the previous data model with newly discovered value in $\mathcal{ID}_p$ and the following data models with all existing values in $\mathcal{ID}_c$ into packet sequences. Detailed design of the algorithm to calculate `Value` is introduced in the relation learning module. The bigger `Value` in the triple, the higher probability $\mathcal{ID}_c$ will be mutated when $\mathcal{ID}_p$ has been mutated in the previous packet.

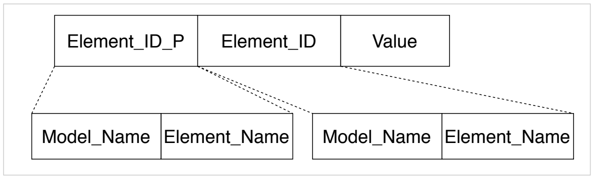

Fig. 5. Cell structure in Relation Table.

### B. Relation Learning

To construct the relation table in Section IV-A, PAVFuzz adopts a lightweight relation learning algorithm to quantify the relations between elements in different states. This module is triggered when a packet with mutated elements achieves new code coverage.

```
Algorithm 2: Relation Learning
   Input: $\mathcal{D}$: Current data model to generate packets
   Input: $\mathcal{D}'$: Next data model
   Input: $\mathcal{S}_{id}$: Set of elements mutated in $D$
   Output: $\mathcal{T}_r$: Relation Table
1  $\mathcal{S}_{id} \leftarrow$ ElementPruning($\mathcal{S}_{id}, \mathcal{D}$)
2  for $elem \in \mathcal{S}_{id}$ do
3      $m \leftarrow$ CurPacketRegeneration($\mathcal{D}, elem$)
4      $\mathcal{S'}_{id} \leftarrow$ AnalyzeDataModel($\mathcal{D}'$)
5      for $elem' \in \mathcal{S'}_{id}$ do
6          $tempvalue \leftarrow 0$
7          $\mathcal{M}' \leftarrow$ CollectPackets($\mathcal{D}', elem'$)
8          for $m' \in \mathcal{M}'$ do
9              $seq \leftarrow$ GeneratePacketSeq($m, m'$)
10             $res \leftarrow$ InjectToEUT($seq$)
11             if CheckRes($res$) $= NewCov$ then
12                 $tempvalue \leftarrow tempvalue + 1$
13         $\mathcal{T}_r \leftarrow$ UpdateTab($elem, elem', tempvalue$)
14 return $\mathcal{T}_r$
```

Algorithm 2 illustrates with details how PAVFuzz learns the relations between elements. The algorithm starts with three inputs: 1) current data model $\mathcal{D}$ whose corresponding generated packet triggers the algorithm, 2) the next data model $\mathcal{D}'$ which is subsequent to $\mathcal{D}$, and 3) the set of element ids $\mathcal{S}_{id}$ that were just now mutated in $\mathcal{D}$ in this iteration. At the beginning, since not all data elements in $\mathcal{S}_{id}$ contribute to the new code coverage, in order to improve the accuracy of relation learning, PAVFuzz needs to do pruning on $\mathcal{S}_{id}$ (line 1).

During pruning, `PAVFuzz` attempts to remove the mutation on some elements in $\mathcal{S}_{id}$ successively in the order of mutation weight in this iteration and checks whether this packet could still cover the same code as the initial one. If so, the pruning continues until the minimum subset of $\mathcal{S}_{id}$ that is able to achieve the same result. Thereafter, for each element $elem$ in $\mathcal{S}_{id}$, `PAVFuzz` learns the relation between $elem$ and all elements in $\mathcal{D}'$ (lines 2-13). Taking the element in $\mathcal{D}'$ as $elem'$, to calculate the relation between $elem$ and $elem'$, `PAVFuzz` first collects all packets satisfying the following three requirements into set $\mathcal{M}'$: 1) they improved the code coverage before, 2) they were generated based on $\mathcal{D}'$, 3) $elem'$ is one of their mutation elements (line 7). Then it combines the packet $m$ that is re-generated based on each $elem$ in $\mathcal{S}_{id}$ after pruning with each packet $m'$ in $\mathcal{M}'$ into new sequences (line 9). Finally, the sequences are fed into the EUT and `PAVFuzz` checks the execution results. Once a sequence covers new code, the value of relation in corresponding cell of $\mathcal{T}_r$ increases a unit (lines 10-12).

With the fuzzing procedure goes on, more code will be covered and more times of relation learning will be triggered, resulting in the more complete relation table and more high-quality generated packets. This is a positive cycle that increases the overall efficiency when fuzzing those protocols used in autonomous vehicles.

### C. State-sensitive Mutation

With the help of relation table, `PAVFuzz` implements a state-sensitive mutation strategy. Those elements with higher mutation weight are more likely to be mutated with new values in the generated packets. Compared with the traditional element chosen strategy, dynamic mutation weight smartly identifies the key elements in current state, avoiding the situation where large amount of time and computing resources are wasted to mutate elements that are irrelevant.

```
Algorithm 3: State-sensitive Mutation
  Input: $\mathcal{D}$: Current data model to generate packets
  Input: $m_p$: Previous packet sent to EUT
  Input: $\mathcal{T}_r$: Relation Table
  Output: $m$: Output packet after mutation with dynamic
          weight
1  $\mathcal{D}_p \leftarrow$ GetDataModel($m_p$)
2  $\mathcal{S}_{id\_p} \leftarrow$ GetMutationSet($m_p, \mathcal{D}_p$)
3  $\mathcal{S}_{id} \leftarrow$ AnalyzeDataModel($\mathcal{D}$)
4  $Dict \leftarrow \emptyset$
5  for $elem \in \mathcal{S}_{id}$ do
6      $weight \leftarrow 0$
7      for $elem_p \in \mathcal{S}_{id\_p}$ do
8          $weight \leftarrow$
             $weight +$ GetRelationValue($\mathcal{T}_r, elem_p, elem$)
9      $Dict \leftarrow Dict \bigcup < elem, weight >$
10 $\mathcal{S}_{id} \leftarrow$ ChooseByWeight($\mathcal{S}_{id}, Dict$)
11 $m \leftarrow$ Mutation($\mathcal{D}, \mathcal{S}_{id}$)
12 return $m$
```

Detail of the state-insensitive mutation stragety is presented in Algorithm 3. In each iteration, data models of corresponding states are used to serve as the base of mutation and they are orderly traversed by the fuzzer. To calculate the current mutation weight of each element $elem$ in this data model $\mathcal{D}$, `PAVFuzz` needs to refer to the previous packet $m_p$ that was sent to the EUT. `PAVFuzz` analyzes $m_p$, then it gets its data model $\mathcal{D}_p$ and the after-pruning set of mutated elements $\mathcal{S}_{id\_p}$ (lines 1,2). The mutation weight of $elem$ can be calculated with the following formula:

$$W_{elem} = \sum_{elem_p} \mathcal{T}_r < elem_p, elem > \ , \ elem_p \in \mathcal{S}_{id\_p}$$

The calculation sums the value in relation table $\mathcal{T}_r$ with the two sequential indexes $elem_p$ and $elem$, where $elem_p$ means each element in $\mathcal{S}_{id\_p}$ (lines 5-8). The results are kept in an enumerable dictionary whose key is element id and value is the corresponding mutation weight of the element (line 9). Thereafter, `PAVFuzz` chooses the set of elements to be mutated $\mathcal{S}_{id}$ according to the weights of elements in the dictionary (line 10). Finally, a new packet is generated based on the data model $\mathcal{D}$ and the mutation set $\mathcal{S}_{id}$ (line 11).

## V. Evaluation

We implemented `PAVFuzz` using C# and utilized the mutation engine of Peach (community version 3.0.202) [17], which is one of the most widely used protocol fuzzers. To instrument the EUT and collect run-time code coverage information for relation learning, we designed and implemented a compiler wrapper on top of the compiler Clang based on the features of LLVM [14].

We conducted comparative experiments on `PAVFuzz` and other state-of-the-art fuzzers to answer the following research questions:

1) *Is `PAVFuzz` more efficient in fuzzing protocols used in autonomous vehicles than start-of-the-art fuzzers?*
2) *Can `PAVFuzz` effectively expose previously unknown vulnerabilities in those widely used protocols in autonomous vehicles?*

### A. Experiment Setup

We chose three famous in-vehicle protocols, namely RTPS [13], SOME/IP [21], and ZeroMQ [26], as the evaluation subjects. Table I briefly introduces their application in autonomous vehicles. They are all widely used in both academic and industrial community. Therefore, we believe that they are representative for evaluating `PAVFuzz`. Specifically, we chose open-source FastRTPS [5], GENIVI vsomeip [8], and libzmq [25] as the concrete implementations corresponding to these three protocols. Each program is hardened by Google Address Sanitizer [10] when compiled.

TABLE I
Description of Selected Protocols

| Protocol | Description |
|---|---|
| RTPS | RTPS is the standard wire protocol used in Data Distribution Service (DDS [12]), which is adopted in many automatic driving systems such as Adaptive AutoSar [3], Baidu Apollo [2], etc . |
| SOME/IP | SOME/IP is a famous application protocol used in Ethernet-based in-vehicle network systems for service discovery and communication control over ECUs, carmers, radars, and so on. |
| ZeroMQ | ZeroMQ is a lightweight protocol for distributed communication. It has been adopted as alternative protocol in ROS2 (Robot Operating System [20]), a prototype system for automotive driving. |

We compared `PAVFuzz` with two widely used fuzzers: AFL and Peach, which are representatives of mutation-based and generation-based fuzzers respectively. They have been able to uncover security flaws in a large range of real-world protocol programs. For comparison, we used the branch coverage achieved and the number of unique bugs detected as metrics. The first metric is commonly used to measure the effectiveness of fuzzers while the second metric indicates the ability to detect vulnerabilities.

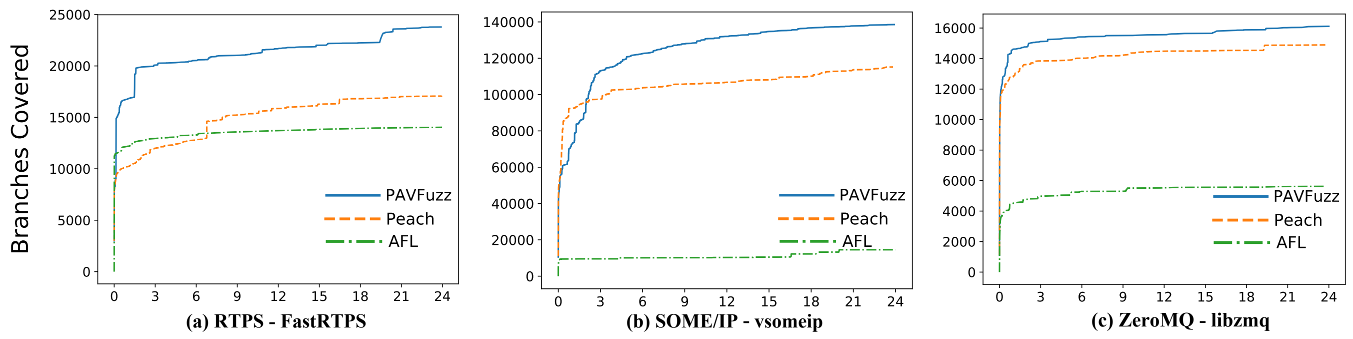

Fig. 6. Average number of code branches achieved by different fuzzing tools in 24 hours.

## B. Efficiency of Fuzzing

To assess the efficiency, we compared the covered branches within the same time. The more branches a fuzzer covers, the more efficient it is. We ran each experiment for 24 hours and repeated each experiment for 5 times to establish statistical significance of results. Detailed results of each protocol are presented in Figure 6 and the overall improvements are summarized in Table II, where $\mathcal{I}_{\mathcal{A}}$ and $\mathcal{I}_{\mathcal{P}}$ indicate the improvement compared to AFL and Peach respectively.

TABLE II
AVERAGE NUMBER OF CODE BRANCHES ACHIEVED BY EACH FUZZER WITHIN 24 HOURS.

| Subject | AFL | Peach | PAVFuzz | $\mathcal{I}_{\mathcal{A}}$ | $\mathcal{I}_{\mathcal{P}}$ |
|---|---|---|---|---|---|
| FastRTPS | 14034 | 17107 | 23784 | +69.47% | +39.03% |
| vsomeip | 14562 | 115147 | 138548 | +851.44% | +20.32% |
| libzmq | 5621 | 14894 | 16114 | +186.67% | +8.19% |
| **AVERAGE** | | | | +369.19% | +22.51% |

From Figure 6, we can observe that at the beginning of each experiment, all the three fuzzers covered new code branches at rapid speed. Then, AFL first slowed down, gradually bogged down and finally reached a saturation state where the improvement of branch coverage turned extremely hard for it. It demonstrates that for mutation-based fuzzers, without the help of data models for under-test protocols, their bit/byte level mutations over network packets are not efficient as they tend to destroy the legal structure of packets. Therefore, AFL finally achieved the least branch coverage in the selected three in-vehicle protocols. As for Peach, it performed better than AFL with the help of user-provided data models of each state. However, the coverage improvement became slower and slower. `PAVFuzz` performed the best and achieved the highest branch coverage in all of the three protocols. Because of the novel relation learning and state-sensitive mutation with dynamic weight, `PAVFuzz` kept a faster speed of branch improvement. Specifically, compared with Peach and AFL, `PAVFuzz` increases branch coverage by averagely 22.51% and 369.19% within 24 hours.

## C. Previous Unknown Vulnerabilities

Besides efficiently improving branch coverage, `PAVFuzz` has also exposed 12 serious previously unknown vulnerabilities in these three protocols, while AFL and Peach only detected 1 and 6 of them respectively. These vulnerabilities have been confirmed and may cause serious hazards. We present these vulnerabilities in Table III.

Figure 7 illustrates a heap-buffer-overflow vulnerability exposed by `PAVFuzz` in vsomeip. This bug occurs in function `look_ahead`, where the protocol wants to deserialise a field from the packet stream. The bug is triggered in line 141 and the variable `position` is a global iterator of the packet. The developers made a mistake to merely check that `_index` does not exceed the limit in line 138-139. Once the result of `position` add `_index` exceeds the size limit, the heap-buffer-overflow bug occurs. Figure 8 presents a stack-buffer-overflow vulnerability exposed by `PAVFuzz` in FastRTPS. The cause

TABLE III
STATISTICS ON PREVIOUSLY UNKNOWN BUGS EXPOSED BY PAVFUZZ.

| Subject | Vulnerability | AFL | Peach | PAVFuzz |
|---|---|---|---|---|
| FastRTPS | stack-buffer-overflow-1 | ✗ | ✓ | ✓ |
| | stack-buffer-overflow-2 | ✗ | ✓ | ✓ |
| | stack-buffer-overflow-3 | ✗ | ✗ | ✓ |
| | stack-buffer-overflow-4 | ✗ | ✓ | ✓ |
| | stack-buffer-overflow-5 | ✗ | ✓ | ✓ |
| | stack-buffer-overflow-6 | ✗ | ✗ | ✓ |
| | heap-buffer-overflow-1 | ✗ | ✗ | ✓ |
| | heap-buffer-overflow-2 | ✗ | ✗ | ✓ |
| | heap-buffer-overflow-3 | ✗ | ✗ | ✓ |
| vsomeip | allocate-out-of-memory | ✗ | ✓ | ✓ |
| | heap-buffer-overflow | ✗ | ✗ | ✓ |
| libzmq | allocate-memory-failure | ✓ | ✓ | ✓ |
| Total | 12 | 1/12 | 6/12 | 12/12 |

```
./implementation/message/src/deserializer.cpp
137 bool deserializer::look_ahead(std::size_t _index,
        uint8_t &_value) const {
138     if (_index >= data_.size())
139         return false;
140
141     _value = *(position_ + static_cast<std::
            vector<byte_t>::difference_type>(_index));
142
143     return _value;
144 }
```

Fig. 7. Previously unknown vulnerability exposed by `PAVFuzz` in vsomeip

```
./include/fastdds/rtps/messages/CDRMessage.hpp
127 inline bool CDRMessage::readDataReversed(CDRMessage_t msg,
        octet* o, uint32_t length)
...
132     for (uint32_t i=0;i < length; i++)
133     {
134         *(o + i) = *(msg->buffer + msg->pos + length -1 -i);
135     }
136     msg->pos += length;
...
183     readDataReversed(mst,dest,8);
```

Fig. 8. Previously unknown vulnerability exposed by `PAVFuzz` in FastRTPS

of this stack-buffer-overflow is similar with the one in Figure 7. It occurs in function `readDataReversed` with the key paramters of `msg` and `length`. However, the developers fail to check whether the address to read in line 134 is legal or not. These buffer overflow vulnerabilities are common in protocols and they are likely to be leveraged to perform attacks such as code execution.

## VI. Related Work

Recently, mutation-based fuzzers [24] have been widely adopted in practice for traditional software testing due to their scalability and efficiency. However, being unaware of file format, trivial mutation-based fuzzers suffer from the difficulty in generating valid inputs and covering large regions of code, especially for those protocol programs that process highly-structured packets. To cope with this, some techniques, such as taint analysis [15] and symbolic execution [22], are applied to explore the program implementation to uncover packet structure. However, due to their problems such as under/over-taint or path exploration [4] when scaled to large program bases, the awareness of the packet format is heavily limited. Our work focuses on optimizing generation-based fuzzers which are capable of providing well-formed packets.

Grammar-based fuzzers generate inputs by leveraging the given context-free grammar thus embrace the capability to produce valid inputs within the grammar model [9]. For example, CSmith [23], a fuzzer designed for C programming language, generates C programs based on randomly selected production rules in the grammar. Radamsa [19] leverages regular and context-free formal languages to represent the structures of input data. However, due to the limitation of context-free grammar, these fuzzers tend to encounter the difficulty in encoding integrity constraints, which are widely used in protocol programs. Moreover, these fuzzers are unaware of the state space of those stateful programs like protocol implementation, thus are mainly utilized to produce inputs for stateless programs (e.g., file processing programs), where no internal state is taken into account.

To better fuzz protocol programs that feature a massive state space, some fuzzers augment additional protocol information to guide packet generation. AFLNet [18] makes automated state model inference by leveraging the server's response codes, which does not scale to those protocols without response code. Pulsar [7], a fuzzer combined with automatic protocol reverse engineering, infers the model for message formats and protocol states by observing the network traffic of the protocol. However, the accuracy of the constructed state model remains a challenge for those complex protocols used in autonomous vehicles. These approaches take the protocol state into consideration when generating new inputs but lose the sight of the relation between different data elements to be generated in different protocol states. Our work takes the relation into account and dynamically adjusts the mutation weight of each data element, biasing the input generation towards a higher probability of maximizing code coverage.

## VII. Conclusion

In this paper, we proposed `PAVFuzz`, a state-sensitive fuzzing framework for securing protocols used in autonomous vehicles. It is equipped with a novel relation learning strategy that is able to automatically learn relations between data elements in different protocol states during fuzzing. Using the learned relations, `PAVFuzz` smartly recognizes the key elements in the data model of each state and chooses them to mutate with dynamic weight. Compared with the random element selection strategy used by traditional generation-based fuzzers, dynamic mutation weight avoids the waste of limited time and computing resources, improving the efficiency of fuzzing greatly. Experiments show that, within the same time budget, `PAVFuzz` covered more code branches than state-of-the-art fuzzers Peach and AFL in selected protocols SOME/IP, RTPS, and ZeroMQ. It has also found 12 serious previously unknown vulnerabilities.

Our future work will mainly focus on the following three aspects: 1) apply and adjust `PAVFuzz` to secure protocols in other in-vehicle network systems, such as CAN; 2) improve the efficiency of `PAVFuzz` by optimizing the mutation operators to better suit in-vehicle protocols; 3) support more features or oracles of in-vehicle protocols such as the vulnerabilities in terms of the real-time feature.

## Acknowledgment

This research is sponsored in part by the NSFC Program (No. 62022046, U1911401, 61802223), National Key Research and Development Project (Grant No. 2019YFB1706200), the Huawei-Tsinghua Trustworthy Research Project (No. 20192000794).

## References

[1] Pedram Amini and Aaron Portnoy. Sulley. 2012.
[2] ApolloAuto. apollo. Website. https://github.com/ApolloAuto/apollo.
[3] Autosar. Adaptive autosar. Website. https://www.autosar.org/standards/adaptive-platform/.
[4] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 2013.
[5] eProsima. Repository of eprosima fastrtps/fastdds project. Website. https://github.com/eProsima/Fast-DDS/.
[6] Fiware.org. Libfuzzer. Website. https://llvm.org/docs/LibFuzzer.html.
[7] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, D. Arp, and K. Rieck. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *SecureComm*, 2015.
[8] GENIVI. vsomeip. Website. https://github.com/GENIVI/vsomeip.
[9] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *PLDI*, 2008.
[10] Google. Google address sanitizer. Website. https://github.com/google/sanitizers/tree/master/address-sanitizer.
[11] Andy Greenberg. Hackers remotely kill a jeep on the highway—with me in it. Website. https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/.
[12] Object Management Group. Data distribution service. Website. https://www.omg.org/spec/DDS/1.4/.
[13] Object Management Group. Dds interoperability wire protocol. Website. https://www.omg.org/spec/DDSI-RTPS.
[14] llvm.org. Clang: a c language family frontend for llvm. Website. http://clang.llvm.org/.
[15] Zhengxiong Luo, Feilong Zuo, Y. Jiang, J. Gao, Xun Jiao, and J. Sun. Polar: Function code aware fuzz testing of ics protocol. *ACM Trans. Embed. Comput. Syst.*, 18:93:1–93:22, 2019.
[16] Society of Automotive Engineers. Cybersecurity guidebook for cyber-physical vehicle systems. Website. https://www.sae.org/standards/content/j3061_201601/.
[17] PeachTech. Peach fuzzing platform. Website. https://www.peach.tech.
[18] Van-Thuan Pham, Marcel Böhme, and A. Roychoudhury. Aflnet: A greybox fuzzer for network protocols. *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 460–465, 2020.
[19] OUSPG's Protos Genome Project. Radamsa. Website. https://gitlab.com/akihe/radamsa.
[20] ros.org. Ros 2. Website. https://index.ros.org/doc/ros2/.
[21] Dr. Lars Völker. Scalable service-oriented middleware over ip. Website. http://www.some-ip.com/.
[22] Mingzhe Wang, J. Liang, Yuanliang Chen, Y. Jiang, Xun Jiao, H. Liu, X. Zhao, and J. Sun. Safl: Increasing and accelerating testing coverage with symbolic execution and guided fuzzing. *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 61–64, 2018.
[23] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. In *PLDI '11*, 2011.
[24] Michal Zalewski. American fuzzy lop. 2015.
[25] zeromq.org. libzmq. Website. https://github.com/zeromq/libzmq/.
[26] zeromq.org. Zeromq project. Website. https://zeromq.org/.