SPFuzz: Stateful Path based Parallel Fuzzing for Protocols in Autonomous Vehicles

Junze Yu[†], Zhengxiong Luo[✉ †], Fangshangyuan Xia[§], Yanyang Zhao[†], Heyuan Shi[‡], Yu Jiang[✉ †]
[†] KLISS, BNRist, School of Software, Tsinghua University, Beijing, China
[§] School of Computer Science and Engineering, [‡] School of Electronic Information, Central South University

ABSTRACT

Protocols in autonomous vehicles are essential for efficient in-vehicle network communication. To ensure their security, many research efforts have been paid to the fuzz testing of their implementations. However, those fuzzing optimizations often struggle to manage the protocols' complex state, resulting in low efficiency in branch covering and vulnerability detection.

This paper introduces SPFuzz, a stateful path based parallel fuzzing framework to improve the testing performance of protocols in autonomous vehicles. The basic idea is to accelerate fuzzing speed by dividing tasks to reduce conflicts and dispatching them on different fuzzing instances. SPFuzz first leverages protocol state and data models to generate stateful paths, then divides them into discrete tasks and dispatches them based on their complexity and diversity, ensuring a balanced workload distribution across all fuzzing instances. For evaluation, we implement SPFuzz on top of the state-of-the-art protocol fuzzer Peach and conduct experiments on four prominent vehicle protocols, including ZMTP, MQTT, DDS, and DoIP. The results show that, compared to the original parallel mode of Peach, SPFuzz achieves the same code coverage at a speed of 2.8X-473.2X, with 5.52% more branch coverage within 24 hours. SPFuzz uncovered six previously unknown vulnerabilities in those protocol implementations, with five CVEs assigned in the national vulnerability database. Additionally, SPFuzz has been adapted to ECUs from several vendors, such as NISSAN, and triggered a total of four vulnerabilities that may cause system crashes.

1 INTRODUCTION

The rapid evolution of autonomous vehicles necessitates advanced communication protocols to ensure seamless interactions among diverse system components. However, these protocols inadvertently increase the attack surface. Attackers could exploit protocol vulnerabilities for malicious purposes, such as disabling control units or hijacking control rights. For instance, a flaw in the Jeep Uconnect system once allowed attackers to take control of the entire vehicle [18]. Therefore, ensuring the security of protocols in autonomous vehicles is significantly essential.

Fuzzing has emerged as a potent automated vulnerability detection technique for real-world protocol implementations [3, 5, 15]. Prior research primarily focused on improving fuzzer efficiency through innovative algorithms that gather more program information to guide deliberate input modifications [7, 9, 14]. However, the performance improvement achieved by algorithm optimization has its limits. With the advancement of multi-core hardware technology and the abundance of computing resources, another direction to enhancing fuzzing efficiency is to parallel fuzzing tasks.

Yu Jiang and Zhengxiong Luo are the corresponding authors.

There are mainly two types of parallel fuzzers: mutation-based and generation-based. For the original mutation-based fuzzers, such as AFL [20], typically lack explicit task division and allocation. Their reliance on the stochastic nature of mutation often leads to abundant task conflicts in parallel fuzzing. Several optimizations, such as PAFL [8] and AFLteam [10], employ a seed-based division approach to divide the fuzzing task. The collection of the seeds is divided according to different strategies and allocated to various fuzzing instances, which utilize a global view to synchronize information between instances. As for testing protocols in autonomous vehicles, two main issues occur: (i) The lack of protocol structure information often leads to seed mutations that violate message format rules, where a seed is a concatenation of protocol packet sequences in the context of protocol fuzzing. (ii) Since different seeds may refer to identical protocol states, conflicts between fuzzing instances are inevitable. Consequently, designating a singular seed as a mutation-based task falls short in protocol fuzzing. On the other hand, generation-based fuzzers, such as Peach [3], improve performance by leveraging protocol format specifications to generate valid packets. They apportion tasks by fuzzing iterations predicated on the data model, segmenting all iterations of the fuzzing task across different fuzzing instances. While this strategy circumvents conflicts, it is simplistic and inefficient as it sidesteps the protocol state model information of the input program.

For a more accurate and efficient parallel fuzzing framework, we need to solve two main challenges. (i) How to divide tasks to reduce conflicts without information loss? Traditional fuzzers do not take state information into account during task division. However, protocols in autonomous vehicles, which manage communications between various Electronic Control Units (ECUs) and sensors, tend to employ a decentralized structure with multiple states, such as service discovery, service subscription, and publishing. If state models are not accounted for in task division, the tasks may lack specificity and lead to conflicts. (ii) How to efficiently execute parallel fuzzing tasks across all instances? Since different tasks may have different complexities, the task allocation mechanism should be able to balance the workloads across instances. Testing protocols in parallel involves mutually exclusive operating system resources. Effective isolation is crucial during protocol service discovery or publishing stages, where multicast packets are sent to specific addresses and ports, to prevent interference between different fuzzing instances.

This paper introduces SPFuzz, a stateful path based parallel fuzzing framework, to address the above problems. For the first challenge, we use a stateful path based task division mechanism to reduce conflicts and avoid the loss of state information. First, we calculate the operation complexity for each data model by traversing its hierarchical tree structures to obtain the weight of each element. Second, we utilize the diverse state information derived from the autonomous vehicle protocols to construct a weighted

state model. This model explicitly considers the interconnections between the state model and data models. Based on this, we further generate stateful paths with calculated weights, facilitating task division that minimizes conflicts. For the second challenge, we involve efficient parallel execution across all fuzzing instances, aiming to balance workloads and minimize mutual interference. We assess the aggregate score of each candidate stateful path and group paths into tasks with similar weights to optimize task distribution. After these tasks are assigned to different fuzzing instances, SPFuzz employs network namespaces to isolate the task executions, preventing the multicast and broadcast packets generated by different fuzzing instances from affecting each other.

We implement SPFuzz on top of the most widely used protocol fuzzer Peach and evaluate its performance on four prominent protocols, including DoIP, MQTT, DDS, and ZMTP, all of which have been well-tested and widely used in autonomous vehicles. The experimental results indicate that, in comparison to Peach's default parallel mode, SPFuzz achieves identical code coverage at a speed of 2.8X-473.2X while gaining 5.52% more branches on average over 24 hours. Meanwhile, SPFuzz exposed six previously unknown bugs in these implementations, with five CVEs assigned in the national vulnerability database. SPFuzz has also been adapted to test the ECUs used in real vehicle devices from several vendors, such as NISSAN, and discovered four bugs that can cause ECU crash or unresponsive, highlighting the practicality of SPFuzz.

2 BACKGROUND

Protocols in Autonomous Vehicles. Autonomous vehicles rely on networks like the Controller Area Network (CAN) to connect sensors and ECUs, which facilitates crucial data collection and control of vehicle components. As data demands and speed requirements have increased, alternative networks such as LIN, FlexRay, and MOST have been developed to meet these evolving needs.

Modern autonomous vehicles increasingly adopt 100BASE-T1 automotive Ethernet, leveraging the established TCP/IP protocol framework for highly efficient data transmission. It enables advanced communication middleware, like the Data Distribution Service(DDS) and ZMTP, enhancing both intra- and inter-vehicle communications. While these advancements significantly improve vehicle networking, they also underscore the importance of eliminating potential security bugs in their implementations.

Protocol Fuzzing. Fuzzing is a crucial technique for detecting flaws in network protocol implementations. It can be categorized into two types based on the packet production method: mutation-based and generation-based.

Mutation-based fuzzers operate by randomly mutating existing inputs from a defined corpus, eliminating the need for prior knowledge of protocol specifications, thus simplifying use [14, 20]. These fuzzers adapt to protocol implementations by using workarounds, such as crafting test harnesses for unit testing specific server states or treating inputs sent to the server as concatenated messages for system testing. Meanwhile, they are typically enhanced with a feedback loop, e.g., code coverage, for optimization. However, they may encounter hurdles with highly structured packets due to a lack of format specifications and are not scalable in black-box scenarios because of limited system insight.

In contrast, generation-based fuzzers generate packets using specified protocol models, encompassing data and state models, making them more suitable for protocol fuzzing [3, 5, 15]. The state model, typically represented as a graph, outlines valid message sequences during server interactions, while the data model defines the acceptable message format for each state, including field types, sizes, and valid value ranges. Adapting these fuzzers requires understanding the protocol's interaction logic, which users must obtain by analyzing source code or reviewing protocol specifications. Their structured approach provides a more organized and logical method for scrutinizing protocol implementations, making them a preferred choice.

3 SYSTEM DESIGN

Figure 1 shows an overview of SPFuzz, which follows a parallel fuzzing process that takes the same input as traditional generation-based fuzzers. The input includes the target protocol implementation and protocol specification that is represented by the data models and state models. The data model details the syntactic and semantic information necessary for generating data packets, while the state model depicts the execution process of a fuzzing task, reflecting the state changes of the system under test.

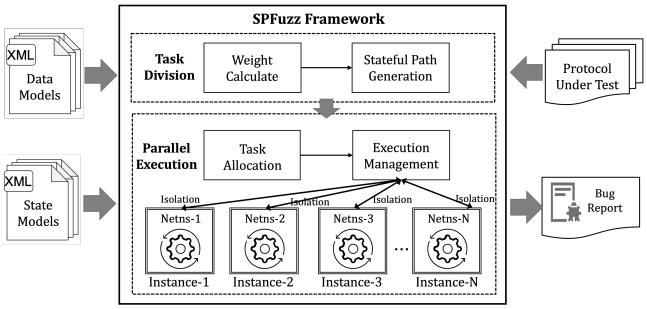

Figure 1: SPFuzz Overview. It mainly consists of two components: (i) a task division part to generate a stateful path based on the protocol state and data model; (ii) a parallel execution part to allocate tasks, and manage isolation execution.

Given the data and state models, instead of starting fuzzing and generating data packets directly, SPFuzz thoroughly analyzes and computes these models first. This is instrumental in facilitating a logical division of tasks based on the insights derived from the models' information. Initially, it computes the mutation complexity inherent in each data model used for fuzz testing. This computation involves a systematic traversal of the data model's elements, which are organized in a hierarchical tree structure. Subsequently, weights are methodically assigned to the state nodes according to the interrelations between the state and data models. This forms the basis for generating various stateful paths derived from the weighted state model. Distinct path sets are delineated in the state model, with each path's weight calculated. A weight-based task allocation module combines the path sets into tasks of similar weights, thereby balancing the computational workloads. Finally, these tasks are allocated to different fuzzing instances for testing, which initiate the fuzzing loop. During parallel execution, isolation techniques are employed to ensure that the tested protocol processes the generated

data packets independently, preventing mutual interference. This module facilitates efficient parallel fuzz testing.

3.1 Task Division

3.1.1 Weight Calculation. Protocol packets, characterized by their high structural organization, are represented by a data model. This model, resembling a tree structure, comprises nodes of diverse elements detailing their hierarchical arrangement, types, attributes, and inter-element relationships. In the fuzzing process, each element type is associated with the specific set of mutators that operate based on the element's attributes, resulting in diverse data packets. The data model determines the computational complexity, referred to as its weight, of a network packet type during fuzzing.

This module computes the weight for all data models B by traversing their tree structure. For each mutable element node, the module retrieves relevant mutators based on the node's type and calculates the count of executable operations for each mutator, considering the node's specific attributes. The aggregate of these operations constitutes the element's weight. If the element possesses dependencies, like $B814$ or $2>D=C$ relationship, with other elements, it is omitted from the calculation. The weights of all child elements are sequentially calculated and then multiplied to determine the total weight of these sub-tree. This total is subsequently added to the parent element's weight, yielding the overall weight of their subtree. The module recursively calculates, moving from the bottom upwards, to determine the weight of this specific data model. Subsequently, it computes the corresponding weight for each data model associated with every state in the state model.

3.1.2 Stateful Path Generation. Following the completion of the weight calculations across all data models, we obtain a weighted state model manifesting as a weighted directed graph. This module methodically generates a set of paths informed by the state model's information, specifically for the allocation of fuzz testing tasks.

The algorithm 1 details how SPFuzz generates the set of stateful paths. The algorithm starts with two primary inputs: 1) Weighted Data Models $\mathcal{WDB}_{\mathcal{M}}$, derived from preceding calculations, and 2) the State Model $\mathcal{S}_{\mathcal{M}}$, characterized as a directed graph. We begin the process by identifying the initial state node $(COC4_{\mathcal{I}}$ within the state model. Following this, we engage in path generation through the recursive Generate procedure, which takes the current state $(COC4_{\mathcal{T}}$, the path $\mathcal{P}$, and the path's weight $,_{\mathcal{P}}$ as parameters (line 4). Each state in the model encompasses a variety of actions. We commence by determining the type of each action (lines 6-8). If the Action $\mathcal{T}$ is of either $DC?DC$ or $=?DC$ type, we append it to the Path and procure its corresponding Weighted Data Model $\mathcal{WD}_{\mathcal{M}}$ (lines 9-11). Then, we integrate the weights $\mathcal{W}_{\mathcal{D}}$ of the data model $\mathcal{D}_{\mathcal{M}}$ corresponding to all $DC?DC$ actions into the overall path weight $\mathcal{W}_{\mathcal{P}}$. Subsequently, we assimilate the weights $\mathcal{W}_{\mathcal{D}}$ of the data model $\mathcal{D}_{\mathcal{M}}$ associated with the $DC?DC$ action into the cumulative path weight $\mathcal{W}_{\mathcal{P}}$ (lines 12-14). Conversely, if the Action $\mathcal{T}$ is of the $O=64(COC4$ type, we include the current path $\mathcal{P}$ in the $(4C_{\%OC}$ and recursively invoke the Generate procedure to construct the path for the subsequent state (lines 15-18). In instances where the action type is identified as $(C>?$, which indicates a termination state, we add the current Path to the Path Set, thereby marking the completion of this process segment (lines 19-21).

This procedure iterates until it has traversed all states within the state model, effectively dividing it into distinct state paths. These paths are then allocated to different fuzzing instances. In generating paths, the algorithm utilizes state and data models' information to prevent overlap. Additionally, the weight of each path is calculated to balance the workload of the allocated tasks.

Algorithm 1: Stateful Path Generation

Input: $\mathcal{WDB}_{\mathcal{M}}$: Weighted Data Models
Input: $\mathcal{S}_{\mathcal{M}}$: State Model
Output: $(4C_{\%OC}$: The set of stateful paths

```
1  Algorithm
2      (4C%OC ← ∅, P ← ∅
3      (COC4_I ← GetInitState(S_M)
4      Generate( (COC4_I• ∅• 0)
5  Procedure Generate( (COC4_T• P• , P)
6      2C8>=B ← GetActions((COC4_T)
7      for T ∈ 2C8>=B do
8          switch T") ~?4 do
9              case $DC?DC ∨ =?DC do
10                 D_M ← GetDataModel( T)
11                 P ← Joint(P• T)
12                 if Equal( T") ~?4• $DC?DC) then
13                     , D ← GetWeight(WDB_M• D_M)
14                     , P ← , P + , D
15             case O=64(COC4 do
16                 for (COC4_N ∈ T") OA64C(COC4B do
17                     (4C%OC ← (4C%OC ∪ {P• , P}
18                     Generate( (COC4_N• P• , P)
19             case (C>? do
20                 (4C%OC ← (4C%OC ∪ {P• , P}
21                 return
22     return
```

3.2 Parallel Execution

3.2.1 Weight-Aware Task Allocation. The module effectively allocates tasks to fuzzing instances based on weighted stateful paths, aiming for workload balance. Algorithm 2 provides our weight-aware task allocation strategy.

The algorithm 2 outlines the task allocation process of SPFuzz, which assigns tasks to fuzzing instances based on the weight of stateful paths. Given the stateful path set $(4C_{\%OC}$ and the number of fuzzing instances $\mathcal{N}_5$, we initially aggregate the weights of all paths in $(4C_{\%OC}$. This total is then divided by the number of fuzzing instances $\mathcal{N}_5$, to establish an ideal average weight $\mathcal{W}_{OE6}$ for each task (line 2). Following this, tasks are sequentially allocated to each fuzzing instance (lines 3-9). The algorithm employs the procedure GetNextPath to select the next candidate path from the unallocated Path $(4C_{\%OC}$ for a given set of tasks (line 6). It initializes $(2>A4$ and $\mathcal{P}_{=4GC}$ for tracking the candidate (line 12). For each candidate path $\mathcal{P}_2$ in the unallocated set, the procedure calls CalcScore to calculate its score based on the similarity to each path in the current $(4C)_{OB:}$, where a higher homogeneity results in a higher score. These scores are then aggregated to determine the most

Algorithm 2: Weight-Aware Task Allocation

```
Input: (4G%0C : The set of stateful paths
Input: N5 : The number of fuzzing instances
Output: )0B:B: The array of tasks allocated to each instances
1  Algorithm
2      W0E6  SumWeight¹(4G%0C º•N5
3      for 8 from 0 to N5 do
4          )0B:B»8¼ ; ,W)  0
5          while W) Ÿ W0E6 do
6              P=4GC  GetNextPath()0B:B»8¼(4G%0C )
7              W)  W ) , P=4GC"F486 C
8              )0B:B»8¼ )0B:B»8¼ [ P=4GC
9              (4G%0C  (4G%0C n P=4GC
10     return )0B:B
11 Procedure GetNextPath((4G)0B: •(4G%0C )
12     P=4GC ; ,(2>A4N  0
13     for P2 2 (4G%0C do
14         (P  0
15         for PC 2 (4G)0B: do
16             (P  (P , CalcScore¹P2•PCº
17         if (2>A4P ¡ (2>A4N then
18             (2>A4N  (2>A4P , P=4GC P 2
19     return P=4GC
```

suitable candidate path for addition to the task set (lines 13-16). If the current Path's score ($(_P$) is greater than the current highest score, the path P_2 becomes the new candidate and the $(2>A4_N$ is updated accordingly (lines 17-18). The procedure returns the path $P_{=4GC}$ with the highest score, which is then added to the task set of the current fuzzing instance (line 19).

Once a candidate path is chosen, it is added to the task, and the algorithm updates both the task's total weight and the status of the unallocated path set (lines 7-9), continuing this process until the total weight of the task approaches the average weight W_{0E6} (line 5). This iterative approach guarantees that the task allocation for each instance aligns closely with the average weight, thus facilitating a balanced distribution of the workloads.

3.2.2 Execution Management. Parallel execution e ciently utilizes multi-core computational resources. However, unlike fuzz testing of binary library objects, testing protocol implementations in parallel involves operating system resources, such as IP addresses and ports. Modifying the con guration les of the program under test can enable the use of independent resources, but this may introduce inconveniences in testing. Moreover, during protocol service discovery or publishing stages, multicast or broadcast network packets are generated and sent to speci c addresses. Without e ective isolation, packets generated by one instance could impact others' SUT. The parallel execution module addresses this by establishing separate network namespaces for each SUT prior to execution, ensuring that each has its own independent protocol stack and networking space. It creates network interface devices and activates them within these namespaces. After setting up the execution environment, the module associates the fuzzing instance and the SUT with the designated network namespace (Netns). This approach not only isolates the instances from each other but also proves to be resource-e cient and lightweight.

Table 1: Average number of code branches achieved by each fuzzer within 24 hours.

Subject	Peach-P	SPFuzz	Time $_P$	Time $_S$	$I_{1A0=2}$	Speed
libzmq	8039	8974	14197s	30s	+11.63%	473.2X
NanoMQ	9025	9450	78678s	198s	+4.71%	397.4X
CycloneDDS	22698	23155	9502s	3371s	+2.01%	2.8X
libdoip	199	208	1423s	11s	+4.52%	129.3X
AVERAGE					+5.52%	250.6X

4 EVALUATION

We have implemented SPFuzz on the most widely used generation-based protocol fuzzer Peach (community version 3.0.28) [32]. The Task Division module is implemented using the Python Document Object Model API. It is utilized to parse the Data Model as a tree for calculating the weights of elements and to interpret the state model as a weighted graph for stateful path generation. In the Parallel Execution module, we added a new task allocation strategy for Peach. We utilize the networking 8? tool to establish Netns and network interfaces for each instance, thereby isolating the execution of the allocated task within their respective spaces. For evaluation, we answer the following three research questions:

- RQ1 Is SPFuzz more e cient than Peach's default parallel mode in fuzzing protocols used in autonomous vehicles?
- RQ2 Can SPFuzz e ectively expose previously unknown vulnerabilities in protocol implementations?
- RQ3 How does SPFuzz's performance when adapted to real-world autonomous vehicles?

Subjects and Experiment Settings. We selected four in-vehicle protocols: ZMTP, MQTT, DDS, and DoIP, which are widely used in autonomous vehicles. We used their corresponding popular open-source implementations [1, 2, 4, 21] as the under-test subjects. Meanwhile, we used Clang to insert trace-pc-guard instrumentation, a feature in LLVM SanitizerCoverage [19], to collect branch coverage, and enabled ASan and UBSan [17] to harden the under-test protocol programs. We evaluate SPFuzz's performance by comparing it with Peach's default parallel mode by setting four parallel instances. We initialized SPFuzz and Peach with the same Pit con guration les that specify the data and state models of each protocol.

Metrics. We employed three metrics for our evaluation: (i) branch coverage achieved, (ii) the duration SPFuzz requires to reach the maximum branch coverage achieved by Peach's parallel fuzzing within 24 hours, and (iii) the number of unique bugs detected. The rst metric is commonly used to measure the e ectiveness of fuzzers, the second metric assesses the parallel fuzzing speed, and the third metric indicates vulnerability detection capabilities.

4.1 Coverage Analysis

We ran each tool in parallel mode with four fuzzing instances for 24 hours and repeated each experiment ve times to establish the statistical signi cance of the results. Detailed results are presented in Figure 2, with overall improvements summarized in Table 1, where $)8<4_P$ indicates the time that Peach used to achieve the maximum

(a) ZMTP-libzmq (b) MQTT-Nanomq (c) DDS-CycloneDDS (d) DoIP-libdoip

Figure 2: Average number of code branches achieved by SPFuzz and Peach's default parallel mode within 24 hours.

branch coverage). T_{S} indicates the time taken by SPFuzz to achieve the same number of covered branches, and $\Delta T_{0=2}$ indicates the coverage improvements compared to Peach.

Figure 2 demonstrates that, in all of the four projects, SPFuzz achieves the highest branch coverage. We can observe that both fuzzers rapidly covered new code branches at the beginning of each experiment. Then, Peach first slowed down, gradually bogged down, and finally reached a saturation state where the improvement of branch coverage turned extremely hard for it. SPFuzz kept a faster speed of branch improvement. Due to libzmq's diverse messaging patterns, including REQ-RSP, SUB-PUB, and PIPELINE, which result in more complex states, SPFuzz shows a slightly greater coverage improvement of 11.63% on this program. Conversely, on CycloneDDS, where service discovery is limited to two modes-unicast and multicast, the overhead of state-sensitive computation offsets the optimization benefits, and the enhancement achieved is relatively minor. Specifically, compared to Peach, SPFuzz achieved an average of 5.52% more branches within 24 hours.

The comparison between T_{P} and T_{S} reflects the fuzzing speed and efficiency. Table 1 shows that SPFuzz achieves the same branch coverage (i.e., the final coverage of Peach) at a speed of 2.8X to 473.2X compared to the original Peach parallel mode. By utilizing protocol state and data models to generate stateful paths and allocate tasks based on them, SPFuzz avoids conflicts, balances workload, and significantly accelerates parallel fuzzing.

4.2 Bug Detection Capability

Besides efficiently improving fuzzing speed, SPFuzz also exposed six serious previously unknown vulnerabilities in the target protocol implementations, while Peach only detected three of them, as shown in Table 2. These vulnerabilities have been confirmed, potentially posing severe hazards. Five of them are assigned CVE identifiers and fixed after a coordinated disclosure.

Table 2: Previously unknown bugs exposed by SPFuzz

Subject	Vulnerability	CVE-ID	Peach-P	SPFuzz
NanoMQ	heap-buffer-overflow-1	CVE-2023-34490	✓	✓
	heap-buffer-overflow-2	CVE in Process	✗	✓
	heap-buffer-overflow-3	CVE-2023-34488	✗	✓
	heap-use-after-free	CVE-2023-34494	✗	✓
	heap-buffer-overflow-4	CVE-2023-34492	✓	✓
libdoip	stack-buffer-overflow	CVE-2024-25188	✓	✓
Total	6	5	3/6	6/6

Case Study. Figure 3 illustrates a heap-buffer-overflow vulnerability exposed by SPFuzz in NanoMQ. This bug occurs in function conn_handler, where the implementation parses the packet fields from the binary stream. The bug is triggered when out-of-bounds accessing the pkt to parse the version field (line 9). The variable pos is an iterator of the packet. The developers incorrectly validated the value of pos due to the incorrectly formulated verified condition between it and the limit max (lines 2-8). Once the sum of *pos and *str_len exceeds the size limit, the heap-buffer-overflow bug occurs. Heap overflow vulnerabilities are critical and likely to be leveraged to perform attacks like remote code execution.

```
1  int32_t conn_handler(uint8_t *pkt, conn_param *cparam, size_t
       max) {
2      if (*str_len > 0) {
3          ...
4          *pos  = (*pos) + (*str_len);
5      }
6      ...
7      rv = (len_of_str < 0  pos + 4 > max) ? PROTOCOL_ERROR : 0;
8      if (rv != 0) return rv;
9      cparam->pro_ver = pkt[pos]; // Out-of-bounds memory access
10     pos++;
11     ...
12 }
```

Figure 3: The CVE-2023-34488 exposed by SPFuzz in NanoMQ.

4.3 Real Device Testing

We also adapted SPFuzz to real autonomous vehicles to evaluate its performance in real-world scenarios. Bluetooth protocol is widely used in autonomous vehicles for communication between the ECU and the user's devices. We adapted SPFuzz to fuzz the Bluetooth protocol of the ECU used in autonomous vehicles and Figure 4 shows details of the real adaptation.

In the example, we started three fuzzing instances in parallel, with each instance connected to a single transmitter to support parallel fuzzing. The under-test ECU processes the fuzzed packets received by the connected wireless terminal. The three parallel fuzzing instances were initiated with the same protocol model but were allocated different tasks based on different stateful paths under SPFuzz framework, as reflected in the visualized workload curves of these three instances.

We adapted SPFuzz to real ECUs from several vendors and triggered a total of four vulnerabilities, as shown in Table 3. We tested three Bluetooth protocols, AVRCP, AVDTP, and HFP, and all the selected ECUs support all three Bluetooth protocols. Due to the black-box nature of the ECUs, we cannot determine the root cause of the vulnerabilities and only provide the symptoms. These critical