A Comprehensive Study on Code Clones in Automated Driving Software

Ran Mo, Yingjie Jiang, Wenjing Zhan, Dongyu Wang, Zengyang Li
School of Computer Science and Hubei Provincial Key Laboratory of Artificial Intelligence and Smart Learning
Central China Normal University, China
moran@ccnu.edu.cn, {jyj1212, zwj123, wangdongyu156}@mails.ccnu.edu.cn, zengyangli@ccnu.edu.cn

***Abstract*—With the continuous improvement of artificial intelligence technology, autonomous driving technology has been greatly developed. Hence automated driving software has drawn more and more attention from both researchers and practitioners. Code clone is a commonly used to speed up the development cycle in software development, but many studies have shown that code clones may affect software maintainability. Currently, there is little research investigating code clones in automated driving software. To bridge this gap, we conduct a comprehensive experience study on the code clones in automated driving software. Through the analysis of *Apollo* and *Autoware*, we have presented that code clones are prevalent in automated driving software. about 30% of code lines are involved in code clones and more than 50% of files contain code clones. Moreover, a notable portion of these code clones has caused bugs and co-modifications. Due to the high complexity of autonomous driving, the automated driving software is often designed to be modular, with each module responsible for a single task. When considering each module individually, we have found that *Perception*, *Planning*, *Canbus*, and *Sensing* modules are more likely to encounter code clones, and more likely to have bug-prone and co-modified clones. Finally, we have shown that there exist cross-module clones to propagate bugs and co-modifications in different modules, which undermine the software's modularity.**

***Index Terms*—Automated Driving Software, Code Clone, Co-modification, Bug-proneness, Software Modularity**

I. Introduction

In recent years, autonomous driving has become one of the hottest topics, and autonomous vehicles are even regarded as the next potential milestone in artificial intelligence [1]. In automated driving software, the complex driving task is often implemented by a combination of multiple modules, such as *localization*, *mapping*, *planning*, *perception*, *decision*, and *prediction* modules, etc. In this way, each module will have its own task, which reduces the complexity of autonomous driving by dividing it into a series of more easily solvable problems [2].

In a software system, code clones indicate identical or similar code fragments. Instead of coding from scratch, developers prefer copy–paste or slightly modify existing code fragments to quickly complete other pieces of code, which makes code cloning a common programming practice [3]–[5]. For example, Roy et al. [3] presented that 7% to 23% of cloned code was found in software projects. Tairas et al. presented that 8% of the code lines in JDK 1.4.2 is cloned [6]. Kondo et al. found that only 20.8% of the studied contracts are not a clone of any other contract [4]. For software development and maintenance, code cloning could be both advantageous and harmful. On the one hand, code cloning could improve code reusability and programming productivity [5], [7]. On the other hand, code clones may lead to maintenance difficulties. Numerous studies [8]–[12] have shown that the presence of code clones in a project could negatively impact software maintainability, since the cloning can cause update anomalies in cloned code fragments, and bug propagation as bug fixes need to be applied over all cloned fragments.

Compared with other software systems, automated driving software may even have a higher potential of applying code cloning. Generally, automated driving systems contain multiple different types of sensors and controllers [13], which have different interface standards and parameter settings. Thus, it has a high potential of adopting the form of code cloning to achieve rapid development for autonomous driving systems. For example, in the perception module, it is necessary to read data from multiple sensors (such as RADAR, LiDAR, Cameras, etc.) to monitor the surrounding environment of the vehicle. The initialization and data-retrieving functions of these sensors are similar. Besides, there are also similarities in the functions of reading and calculating vehicle parameters like brake and throttle. In addition, automated driving systems often use or refer to the libraries or patterns of robot operating system (ROS) for their development [14]. Prior work [15] has shown that, when designing and implementing the ROS, the simple way to deal with variability is to copy and paste the code, and then make changes as needed. This is likely to bring code clones [16]. Moreover, due to the complicated and volatile road scenarios, the massive amount of processing data (such as sensor data, map data, and data from other vehicles, etc.), and the high safety standards for driving, automated driving software's test environment and conditions become more complex than others [17]. Code cloning could increase the complexity of the software system, thereby increasing the difficulty and cost of testing [18]. Therefore, there is a demand to have an in-depth understanding of code clones in automated driving software.

However, there is little work investigating the code clones in automated driving software. To bridge this gap, we conducted an empirical study to analyze the code clones in automated driving software from the following perspectives: 1) the existence of code clones, 2) the bug-proneness of code clones, and 3) the co-modifications in code clones. In addition,

to reduce the complexity of auto-driving tasks and software maintenance, automated driving software is often structured into multiple relatively independent modules. Therefore, we further examine 4) the code clones within each of the modules, and 5) the cross-module clones.

Through our experiments on two representative open-source automated driving software, Apollo [19] and Autoware [20], we have found that: 1) Code cloning is prevalent in automated driving software. 31.6% and 35.3% of code lines encounter code cloning for *Apollo* and *Autoware*, respectively; 2) 6.1%-55.9% of code clones in *Apollo* and 5.9%-18.7% of code clones in *Autoware* have been involved in bug fixes, and the average fixing time of bugs related to code clones is much longer than the clone-free bugs; 3) 20.2%-77.2% of code clones in *Apollo* and 8.2%-13.8% of code clones in *Autoware* have caused co-modifications; 4) Code clones are more likely to occur in *Perception*, *Planning*, *Canbus* and *Sensing* modules. Meanwhile, these modules contain the largest amount of bug-prone or co-modified code clones 5) Although automated driving software is often designed to have good modularity, there are still 16.6% cross-module clones in Apollo and 20.2% in Autoware, which bring the implicit couplings between modules. Moreover, some cross-module clones have induced bugs and co-modifications, meaning that they have propagated bugs and changes over different modules, which undermines the independence of modules.

We believe our study has the following contributions:

- To the best of our knowledge, this work is the first study to analyze the code clones in automated driving software. We have provided a fundamental understanding of the existence, bug-proneness, and co-modifications of code clones in automated driving software.
- Specific to the modularity of automated driving software, We have investigated how code clones distribute over different modules, and the impacts of cross-module clones.

The rest of this paper is structured as follows: Section II introduces the basic concepts behind this study. Section III presents the design of our study; Section IV reports our experiments and results; Section V discusses our experimental results; Section VI discusses the threats to validity; Section VII presents research related to our work; Section VIII concludes.

II. Background

A. Automated Driving Software

The automated driving software is an integration of multiple techniques, which are used for completing a series of tasks, including vehicle localization, environment perception, path planning, and motion control [21]. Although there may exist a small difference in the overall architecture of different autonomous driving systems, they often contain the following core tasks:

- **Perception:** Perception is an important task in autonomous driving software, which is responsible for recognizing the environment around the autonomous vehicle.
- **Localization:** To guarantee the safe and reliable operations of autonomous vehicles on the road, accurately locating the position of a vehicle is a prerequisite.
- **Planning:** The planning task is important for driving, which consists of the sub-tasks like path planning, decision making, and motion planning.
- **Control:** After an autonomous vehicle completes self-positioning, perception of the surrounding environment, and planning, this task analyzes the corresponding information and converts it to a series of actions for controlling the vehicle's operations.
- **Human Machine Interaction (HMI)**: This HMI enables a vehicle to communicate with its driver or passengers.

B. Bug-proneness in Code Clones

Code clone is considered as one of the code smells which possibly cause deeper problems in a program. If there exists any bug in an original code fragment, it could spread over all the cloned code fragments [22] since they share similar implementations. Hence, code clones could propagate bugs over different code pieces. In addition, to ensure the driving safety of autonomous vehicles, the tests for the automated driving software usually require specific test scenarios and simulation platform supports [17], resulting in higher cost and more complexity. This makes the detection and fixing of bugs in automated driving software more complicated and challenging. Therefore, understanding the relationship between bug proneness and code clones could facilitate bug fixes for code clones in automated driving software systems.

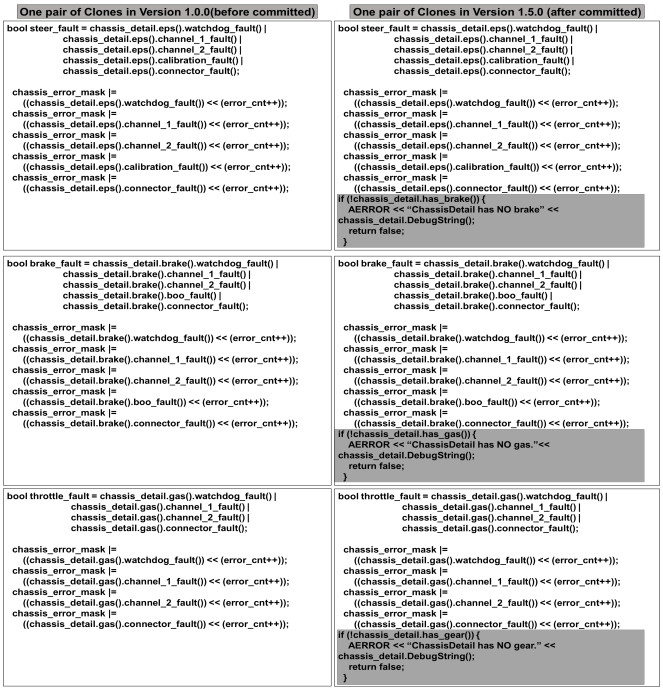

Fig. 1. An example of bug-prone cloned fragments

The sampled code fragments in Figure 1 show a bug-prone code clone from Apollo. The message in its bug-fixing

commit says "*Canbus: fixed bug 6(1) from safety team.*" [23]. This bug fix was made to add the *feedback function* of sensing information to a hardware device for fixing its security flaws. From this example, we can observe that the bug has been propagated over multiple functions through their cloned fragments. To ensure this bug to be repaired completely, developers have to make the same changes in all functions.

C. Co-modified Code Clones

In software development, developers often introduce code clones into software through copy-and-paste operations to shorten the development cycle. When a programmer changes a code fragment, the other cloned code fragments may also need to be changed together consistently to ensure that the software remains consistent [11]. These consistent changes may add additional maintenance costs, and a failure in making such consistent changes is often referred to as a "clone consistency defect", which would negatively affect the software maintainability [12]. In order to explore the consistent changes of clones in the automated driving software, we investigated the clones with co-modifications. In this paper, we refer to code clones with consistent modifications as co-modified clones. Figure 2 shows the example of a co-modified code clone from Project *Apollo*. In version 3.0.0, the three methods share an identical statement for initializing the *acquisition hardware*. In version 3.5.0, the value of the *unit32_tPERIOD* variable needs to be changed, hence all these three similar methods need to be co-modified to keep the consistency.

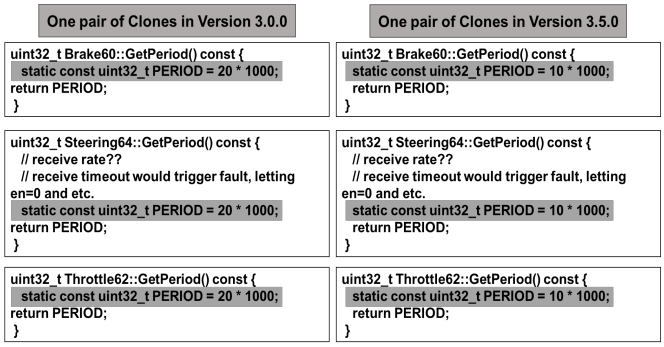

Fig. 2. An example of the co-modified code clone

III. Study Design

A. Objectives and Research Questions

Code cloning is a common implementation for code reuse, which could improve the efficiency and productivity of programming, but code clones can also introduce software quality problems and increase software maintenance costs [24]–[26]. With the advancement of artificial intelligence, autonomous driving has become one of the hottest topics in recent years. Both researchers and practitioners have paid much attention to automated driving systems [1], [2], [27]. However, there is little work studying code clones in automated driving software by far. Therefore, in this paper, we focus the investigations on providing a fundamental understanding of the impact of code clones in automated driving software. To proceed with our study, we attempt to answer the following research questions:

RQ1: To what extent do code clones occur in automated driving software?

–If a large number of code clones exist in automated driving software, meaning code cloning deserves more attention in such software systems. Hence, we examined the existence of code clones by calculating their amounts and lines of code (LOC).

RQ2. Do code clones have a tendency to introduce bugs in automated driving software?

–In this question, we analyzed whether and to what extent code clones bring bugs into automated driving software.

RQ3. Do code clones in automated driving software have co-modifications?

–Here, we explored the maintenance risk posed by code clones. Not all clones in a project would affect maintenance, but there may exist some clones that need to be maintained consistently during the development process. Such synchronous update operations would increase the complexity and cost of maintenance, since whenever a code fragment is changed, developers need to locate its cloned code fragments and make consistent changes. Thus, we investigated the co-modified clones in automated driving software.

RQ4. How do code clones in autonomous driving software distribute over different modules? Which modules have more bug-prone and co-modified clones?

–Since automated driving software is often structured into multiple relatively independent modules, we are interested in exploring which modules developers prefer to apply code clones on. So we further analyzed the distribution of code clones. Meanwhile, to understand how code clones affect the maintenance of related modules, we investigated code clones in which modules are more likely to be bug-prone and co-modified.

RQ5. Do cross-module code clones exist in automated driving software? If so, whether cross-module clones are bug-prone and have co-modifications?

–An automated driving software system is usually divided into multiple relatively independent modules for reducing the complexity of auto-driving tasks and maintenance. If code clones exist across modules, meaning the modules are implicitly coupled by cloned code, which may undermine the independence of modules. Similarly, we also examined the bug-proneness and co-modifications of cross-module clones to explore whether code clones propagate bugs and modifications over different modules.

B. Subjects

We selected the two most representative open-source automated driving projects, Appollo and Autoware, as our subjects. For each project, we downloaded all its stable versions. Table I shows the basic facts. Column "Versions" shows the studied versions of a project. Column "Mod" shows the number of modules of the project varies with its versions, and column

"LOC" means the range of code lines over different versions. Column "File" shows the range of the number of files. The last column shows the number of commits of each project.

TABLE I
STUDIED PROJECTS

Project	*Versions*	*Mod*	*LOC*	*File*	*Commit*
Apollo	1.0.0-8.0.0	13-24	35,791-446,965	262-2,766	19,852
Autoware	1.0.0-1.11.0	7-13	131,222-193,443	557-1,026	3,794

C. Study Procedures

We proceed with our study by following four main steps as shown in Figure 3:

Step 1: *Detecting Code Clones.* Both Apollo and Autoware are mainly written in C++, thus we detect the clones from both header and source files. In this case, we leveraged Simian [28] to detect code clones in the studied projects. Simian [28] is a text-based code clone detection tool and has been widely used for C++ clone detection [29]–[33].We set the parameters by following Ragkhitwetsagul et al. [34] to obtain high recall and precision. The threshold is set to the default value of 6, ignoring the variable name.

Step 2: *Classifying Code Clones.* Given the clone detection results, we 1) calculate the number of clones and their involved lines of code; 2) group them according to their module names; 3) extract the cross-module code clones.

Step 3: *Calculating Clones' Bug-proneness.* We first downloaded all commits from a project's repository. Then we used the keywords to identify bug-fixing commits by referring to the prior study [35]. After that, we extracted the lines changed in each bug-fixing commit, if there is an intersection between the changed lines and the lines from cloned fragments, we consider the corresponding clone to be involved in bugs, and the bug is a clone-related bug. If two or more cloned fragments in a code clone are involved in bugs, we consider this clone tends to bring bugs into the project, and it is a bug-prone clone. In addition, we calculated the bug-fixing time [10] of each clone-related bug. We first used Pydriller [36] to retrieve all commits that foremost "touched" the modified lines of a bug-prone clone, and we call it the bug-inducing commit. Then, we calculated the time span from the bug-inducing commit to the bug-fixing commit as the bug-fixing time. Finally, we computed the average fixing time for all clone-related bugs, and compared it with the one of clone-free bugs.

Step 4: *Identifying Co-modified Clones.* To detect the co-modifications of code clones, we compared code clones of a version, with the clones in its subsequent versions. Assume there is a code clone C_1 in version V_1. If the code clone is also detected in any subsequent version V_i as C_i, then we analyzed the difference between the code fragments of C_1 and C_i. If the code fragments from different clone pieces have changed the same lines, we then consider this clone has undergone co-modifications. To obtain enough co-modified clones, we used four subsequent versions for the comparisons. We have removed duplicated clones to avoid biases. Following the above procedures, we can get all code clones involved in co-modifications.

We have shared the detection results and support tool on Github: *https://github.com/vvioletNego/ccParser*

IV. EXPERIMENTAL RESULTS

RQ1: To what extent do code clones occur in automated driving software?

To answer this question, we detected code clones from each project and calculated the LOC of the cloned fragments. Table II and III summarize the experimental results. Column "Version" shows a version number. Column "NC" shows the number of code clones identified in this version. Column "SLOC" indicates the total lines of code in the version. Column "PLOC" indicates the ratio of the LOC of cloned fragments to the total LOC. Column "NF" indicates the number of files with code clones, and Column "PNF" presents the proportion of files with code clones.

Using version 1.0.0 of Apollo in Table II as an example, we can observe that this version contains 290 code clones with 35,791 LOC. About 33.2% of all LOC are involved in code cloning. Besides, 137 files have code clones, which is 52.3% of all files.

Considering all versions of the two projects together, we can see that 24.6% - 38.8% of LOC are involved in code clones, and more than 50% of files contain code clones. These results indicate that code clones commonly happen in automated driving software, which deserve more investigation.

Answer to RQ1: Code clones commonly happen in automated driving software, with an average of 31.6% and 35.3% lines being cloned fragments in Apollo and Autoware, respectively. More than 50% of the files in both projects contain code clones.

TABLE II
STATISTICS OF CODE CLONE IN APOLLO

Version	*NC*	*SLOC*	*PLOC*	*NF*	*PNF*
1.0.0	290	35,791	33.2%	137	52.3%
1.5.0	746	93,691	24.6%	388	59.2%
2.0.0	1,401	152,241	26.3%	628	62.9%
2.5.0	1,977	204,234	27.6%	893	67.9%
3.0.0	2,072	218,527	29.3%	990	69.9%
3.5.0	2,944	292,379	29.8%	1,276	66.8%
5.0.0	3,614	349,605	33.7%	1,530	69.3%
5.5.0	4,157	383,345	34.7%	1,670	70.1%
6.0.0	4,431	411,385	35.2%	1,830	70.6%
7.0.0	4,574	424,732	35.9%	1,880	70.4%
8.0.0	4,925	446,965	37.4%	1,988	71.9%
Average	2,830	273,900	31.6%	1,201	66.5%

NC: the number of code clones. *SLOC:* the LOC of code clones. *PLOC:* the ratio of the LOC of cloned fragments to the total LOC. *NF:* the number of files containing code clones. *PNF:* the proportion of files containing code clones

RQ2: Do code clones have a tendency to introduce bugs in automated driving software?

Understanding to what extent bugs happen in cloned code could increase developers' awareness of mitigating the risks from their code cloning activities. Therefore, in this question, we extensively study the relationships between code clones

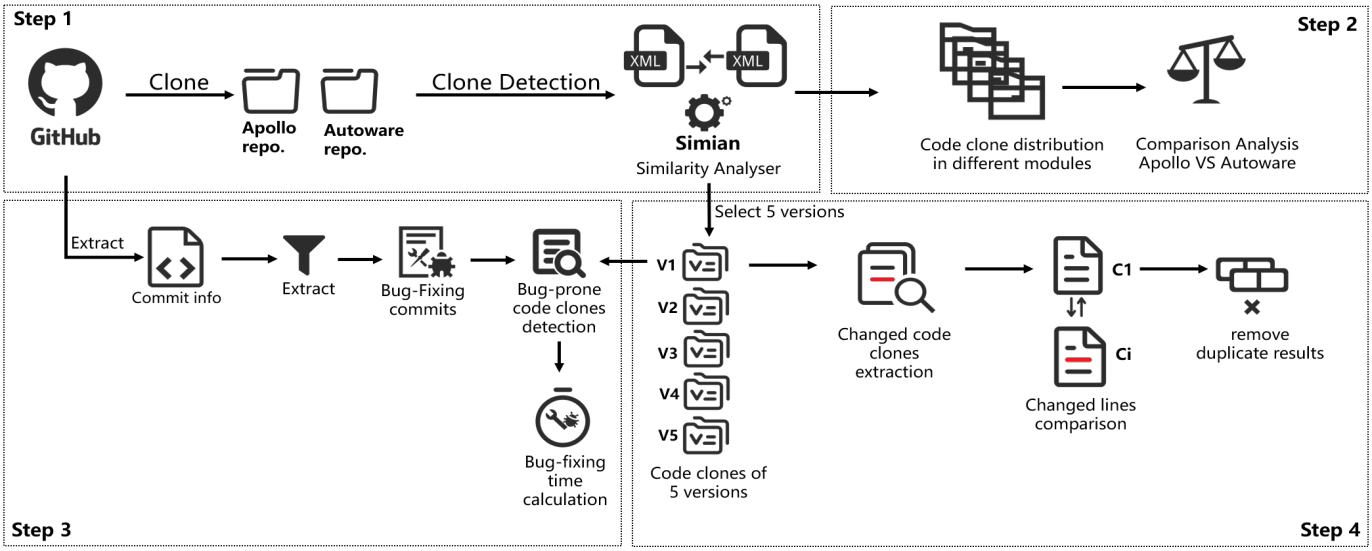

Fig. 3. Experimental Procedures of our Study

TABLE III
STATISTICS OF CODE CLONES IN AUTOWARE

Version	*NC*	*SLOC*	*PLOC*	*NF*	*PNF*
1.0.0	1,354	131,222	33.0%	418	75.0%
1.1.0	1,473	135,911	33.1%	456	77.3%
1.2.0	2,119	176,418	34.8%	552	78.2%
1.3.0	2,313	186,548	35.9%	596	78.7%
1.4.0	2,317	186,932	35.8%	596	78.7%
1.5.0	2,353	190,064	35.6%	613	78.6%
1.6.0	2,473	198,356	35.1%	648	78.4%
1.7.0	2,652	209,069	34.7%	681	77.7%
1.8.0	3,120	240,425	35.9%	744	76.2%
1.9.0	3,209	249,991	35.4%	765	75.4%
1.10.0	3,235	248,723	35.8%	771	75.1%
1.11.0	2,742	193,443	38.8%	572	76.9%
Average	2,447	195,592	35.3%	618	77.2%

and bugs. To carry out the experiments, we extract all bug-fixing commits, and match changed lines in these commits to code clones. When the code fragments in a clone contain buggy lines, we then consider this clone as a bug-prone code clone. Section III-C shows how we identify bug-prone clones in detail.

After obtaining all bug-prone clones, we then calculate their percentage to all clones and the proportion of clone-related bug-fixing commits. Table IV and V report our experimental results. Column "VersionRange" represents the version span. We extracted the bug-fixing commits in the period between the two versions. To guarantee obtaining enough data, we select five versions as the period. Column "NCB" indicates the number of bug-prone clones. Column "PNCB" indicates the proportion of these bug-prone clones. Column "PBC" means the proportion of bug-fixing commits related to cloned fragments. For example, the first row on Table IV shows that 162 code clones (55.9% of all clones) participated in bug-fixing commits. For all the bug-fixing commits, 6.4% of them have changed lines related to cloned fragments. Column "FTCB" represents the average fixing time of clone-related bugs, while Column "FTB" is the average fixing time of clone-free bugs.

For both tables, the proportion of bug fixes related to code clones (PBC) gradually increases as software evolves: from 6.4% to 19.5% in Apollo, from 38.0% to 56.1% in Autoware. It suggests that bugs caused by code cloning continue to increase, and code clones continuously contribute to the software's bug-proneness.

TABLE IV
STATISTICS OF BUG-PRONE CLONES IN APOLLO

VersionRange	*NCB*	*PNCB*	*PBC*	*FTCB*	*FTB*
1.0.0-3.0.0	162	55.9%	6.4%	33 days	28 days
1.5.0-3.5.0	223	29.9%	13.5%	100 days	49 days
2.0.0-5.0.0	240	17.1%	13.6%	136 days	51 days
2.5.0-5.5.0	280	14.2%	11.1%	136 days	51 days
3.0.0-6.0.0	277	13.4%	11.6%	162 days	54 days
3.5.0-7.0.0	160	5.4%	15.5%	150 days	76 days
5.0.0-8.0.0	222	6.1%	19.5%	346 days	186 days
Average	223	20.3%	13.0%	149 days	70 days

NCB: the number of bug-prone clones. *PNCB:* the proportion of bug-prone clones to the total number of clones. *PBC:* the proportion of bug-fixing commits related to code clones. *FTCB:* the average fixing time of clone-related bugs. *FTB:* the average fixing time of clone-free bugs.

After examining the co-existence of code clones and bug-fixing commits, we further analyze the bug-fixing time for bugs that are related to cloned codes or not. This experiment could present whether code cloning has an impact on the time to fix bugs, reflecting how much code cloning affects the maintenance of automated driving software.

Following the prior study [10], we define a bug's fixing time as the time period between the commit that induces it and the commit that fixes it (Section III-C has shown how we calculate bug-fixing time in detail). For both projects, we can observe that the FTCB values are much larger than FTB values, indicating that the fixes of the clone-related bugs

TABLE V
STATISTICS OF BUG-PRONE CLONES IN AUTOWARE

VersionRange	*NCB*	*PNCB*	*PBC*	*FTCB*	*FTB*
1.0.0-1.4.0	80	5.9%	38.0%	161 days	50 days
1.1.0-1.5.0	88	6.0%	36.2%	160 days	64 days
1.2.0-1.6.0	164	7.7%	31.6%	244 days	74 days
1.3.0-1.7.0	187	8.1%	31.9%	352 days	56 days
1.4.0-1.8.0	311	13.4%	35.5%	392 days	66 days
1.5.0-1.9.0	393	16.7%	44.9%	342 days	73 days
1.6.0-1.10.0	330	13.3%	53.2%	380 days	102 days
1.7.0-1.11.0	497	18.7%	56.1%	429 days	112 days
Average	256	11.2%	40.9%	307 days	74 days

are more time-consuming than the clone-free bugs. It means bugs with cloned codes are more difficult to be fixed, and the presence of code clones in automated driving software would hinder their maintenance.

Answer to RQ2: In Apollo, 6.1% - 55.9% of code clones are involved in bugs, and 6.4% to 19.5% bug-fixing commits are related to code clones. In Autoware, the percentages range from 5.9% to 18.7%, and from 31.6% to 56.1%, respectively. Meanwhile, the percentage of clone-related bug fixes increases constantly as software evolves. In addition, the bugs related to code clones are more difficult to be fixed in terms of their fixing time.

RQ3: Do code clones in automated driving software have co-modifications?

For a code clone, if its code fragments are never changed consistently, meaning these fragments indeed are independent in their maintenance, thus we don't need to manage them consistently. But for the cloned code fragments that need to be changed consistently, they could cause maintenance difficulties, since whenever a code fragment is changed, developers have to locate its cloned code fragments and change them as well [37]. Therefore, we investigated whether the code clones have induced co-modifications in automated driving software, and counted the number of modified lines of these co-modified clones.

The experimental results have been summarized in Table VI and VII. Column "NCC" shows the number of co-modified clones. Column "PNCC" shows the proportion of co-modified clones to all code clones. Column "LCC" presents the LOC of co-modified clones, and Column "PLCC" shows the proportion of LOC of co-modified clones to all cloned code lines. Column "CLCC" shows the number of modified lines and Column "PCLCC" represents the ratio of the changed LOC in the co-modifications to the LOC of all co-modified clones. Taking the first row of Table VI as an example, 224 code clones have co-modifications, accounting for 77.2% of all clones. Their lines of code occupy 59.9% of the total LOC of all code clones. For the lines of all co-modified clones, 73.6% of them were modified during the co-modifications.

From Table VI, we can see that, on average, 529 clones are co-modified clones with 18,225 lines of code. The changed LOC in the co-modifications is 8,931, which is 48.9% of all the LOC in co-modified clones.

In addition, we obtain an interesting observation: the proportion of co-modified clones in Apollo decreases from 77.2% to 23.6%, the proportion of lines in co-modified clones decreases from 59.9% to 34.5%, and the proportion of changed lines in co-modifications drops from 73.6% to 38.6% But the number of clones actually increases as software evolves (shown in Table II). This means the consistent changes in clones have been reduced. We further analyzed the revision history of Apollo, and found that co-modified clones had a really high proportion at the early stage of software evolution, but the developers keep changing the code clones to reduce their possibility of requiring co-modifications. It suggests that the developers notice the co-modified clones could induce maintenance difficulties and have applied refactoring actions.

For Autoware, co-modifications still commonly happen in its code clones: 8.2% to 13.8% of clones are co-modified clones, and they contain 10.0% - 19.0% of the total LOC of all clones. During the co-modification, 30.9% -62.4% of lines have been touched.

Answer to RQ3: Co-modifications have commonly happened in code clones in automated driving software. On average, 36.4% of code clones in Apollo and 11.2% in Autoware were co-modified, meaning that code clones do bring maintenance difficulties into the automated driving software.

TABLE VI
STATISTICS OF CO-MODIFIED CLONES IN APOLLO

VersionRange	*NCC*	*PNCC*	*LCC*	*PLCC*	*CLCC*	*PCLCC*
1.0.0-3.0.0	224	77.2%	7,127	59.9%	5,247	73.6%
1.5.0-3.5.0	309	41.4%	9,452	41.1%	5,285	55.9%
2.0.0-5.0.0	510	36.4%	15,583	38.9%	8,536	54.8%
2.5.0-5.5.0	528	26.7%	17,251	30.6%	8,590	49.8%
3.0.0-6.0.0	418	20.2%	14,433	22.5%	7,057	48.9%
3.5.0-7.0.0	863	29.3%	23,298	26.7%	12,108	52.0%
5.0.0-8.0.0	852	23.6%	40,639	34.5%	15,696	38.6%
Average	529	36.4%	18,255	36.3%	8,931	48.9%

NCC: the number of co-modified clones. *PNCC:* the proportion of co-modified clones. *LCC:* the LOC of co-modified clones. *PLCC:* the ratio of LCC to the total LOC of all clones. *CLCC:* the LOC of changed lines in code clones' co-modifications. *PCLCC:* the ratio of CLCC to LCC

TABLE VII
STATISTICS OF CO-MODIFIED CLONES IN AUTOWARE

VersionRange	*NCC*	*PNCC*	*LCC*	*PLCC*	*CLCC*	*PCLCC*
1.0.0-1.4.0	138	10.2%	8,054	18.6%	2,818	35.0%
1.1.0-1.5.0	155	10.5%	7,459	16.6%	2,306	30.9%
1.2.0-1.6.0	265	12.5%	11,627	19.0%	5,296	45.5%
1.3.0-1.7.0	190	8.2%	6,695	10.0%	4,178	62.4%
1.4.0-1.8.0	235	10.1%	7,900	11.8%	4,834	61.2%
1.5.0-1.9.0	306	13.0%	12,184	18.0%	5,848	48.0%
1.6.0-1.10.0	276	11.2%	10,708	15.4%	3,688	34.4%
1.7.0-1.11.0	366	13.8%	12,476	17.2%	4,546	36.4%
Average	241	11.2%	9,638	15.8%	4,189	43.4%

RQ4. How do code clones in autonomous driving software distribute over different modules? Which modules have more bug-prone and co-modified clones?

Due to the significant complexity of automated driving tasks. Automated driving software is often large-scale and complex. To alleviate the burden of software development and maintenance, it is often structured into multiple relatively independent modules, each module will take its own task and have its own design. Hence we further analyze how many modules have code clones and which modules are more likely to get code cloned. In this case, based on the architecture description of the Apollo in the document [38], we manually extracted ten core modules including *Perception*, *Prediction*, *Routing*, etc. Based on the introduction of Autoware's architecture in the document [39], and its directory structure, we extracted eight core modules, including *Perception*, *Planning*, *Sensing*, and so on. According to the determined modules, we can identify code clones in each module.

In order to understand the distribution of code clones in automated driving software, we analyze which modules are more likely to get code cloned. The results have been depicted in Figure 4 and 5, where "PCC" represents the proportion of the number of code clones in a module to all identified clones, and "PLOCC" shows the proportion of the clone lines in a module to the total LOC.

For Project Apollo, more code clones occur in the *Perception* module, but clones in the *Canbus* module occupies a larger amount of LOC. The Perception module has the most clones, but the LOC is not the highest. After reviewing the code fragments, we found that most of its clone fragments are scattered and have fewer code lines. Contrarily, most of the code clones in *Canbus* module cover a large number of code fragments. For Project Autoware, the *Perception* module has the largest number of clones with the most clone lines, reaching 53.8% and 44.2% respectively.

Considering both projects together, we can observe that Perception, Planning, and Canbus(Appollo) or Sensing (Autoware) modules are more likely to encounter code clones. To reveal the possible reasons, we manually reviewed the code in these modules. The *Perception* module is mainly used to monitor the environment. Thus it needs to read data from multiple sensors. Since the initialization and data-retrieving functions of these sensors are similar, code clones happen. The *Planing* module is used to make route planning. It often reuses frameworks or designs, which could cause code clones. Vehicles, computers and sensors can be connected via the Controller Area Network (CAN) bus, Ethernet and/or USB 3.0 [14], [27]. The automated driving system uses similar initialization methods and message transmission methods when controlling brakes, accelerators, and wheel speeds. These sensors and controllers come in a wide variety of models, interfaces, and technical standards. So developers are more likely to use code clones in this module. In Apollo, the *Canbus* module is used to bridge a car and its automated driving software. It collects a vehicle's location information and returns it to the *Control* module. The similarities of processing of location information and parameter setting for brake and throttle have the potential of inducing code clones. Autoware doesn't set *Canbus* as a separate module, instead, it placed the functions of *Canbus* in *Perception* and *Planning* module. The *Sensing* module applies data preprocessing for the raw sensor data. The similarity of data preprocessing functions could bring code clones.

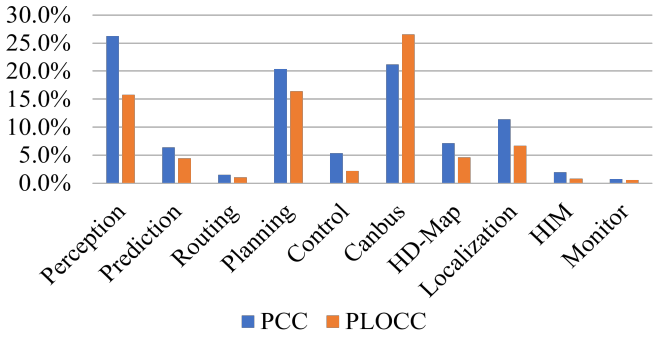

Fig. 4. PCC, PLOCC of core modules in Apollo

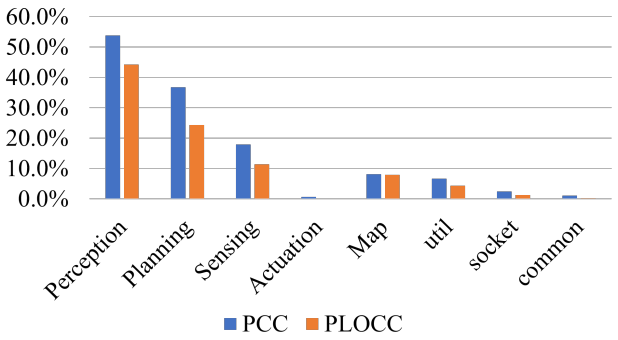

Fig. 5. PCC, PLOCC of core modules in Autoware

Second, we further analyzed the bug-prone clones in different modules to understand which modules are more likely to have bug-prone code clones. Figure 6 and 7 summarize our experiment results. "PBCM" means the percentage of bug-prone clones in a module to all bug-prone clones, and "PLBCM" is the percentage of LOC in bug-prone clones in a module to the total LOC of all bug-prone clones. We can observe that module *Perception*, *Planing*, *Canbus (Apollo)* and *Sensing(Autoware)* have larger PBCM and PLBCM values than other modules, meaning that code clones in these modules are more likely to be involved in bugs. Thus, it is necessary to pay special attention to these modules, especially for their bug fixes.

Finally, we conduct experiments on the co-modified code clones in different modules. The results have been depicted in Figure 8 and 9. "PCMCC" means the ratio of co-modified clones in a module to all co-modified clones, and "PLCMC" means the ratio of co-modified clones' LOC in a module to the LOC of all co-modified clones. "PLMC" indicates the ratio of the changed LOC of the co-modifications within a module to the changed LOC in all co-modifications. We can get the consistent observation: Module *Perception*, *Planning*, *Sensing* and *Canbus* suffer the most from the co-modified clones.

Answer to RQ4: Among the core modules, the modules related to Perception, Planning, Canbus (Apollo), and Sensing

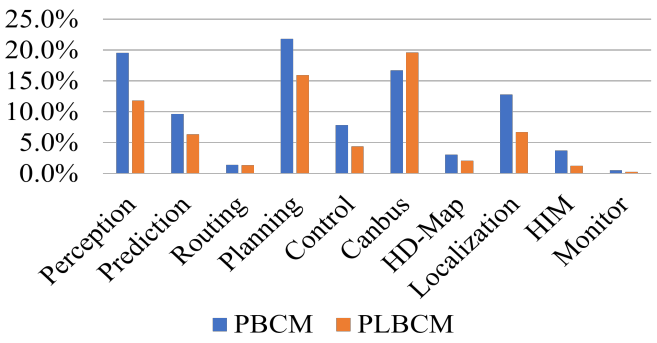

Fig. 6. Distribution of Bug-prone Code Clones in Apollo

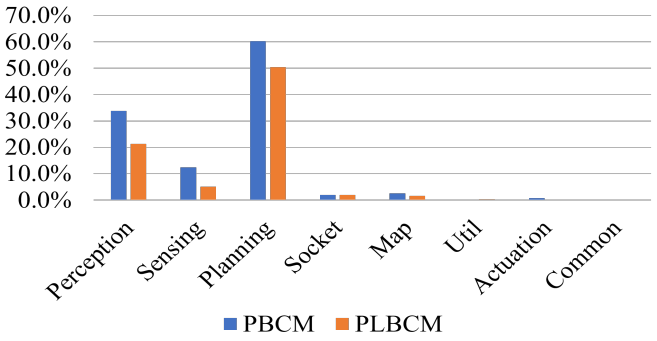

Fig. 7. Distribution of Bug-prone Code Clones in Autoware

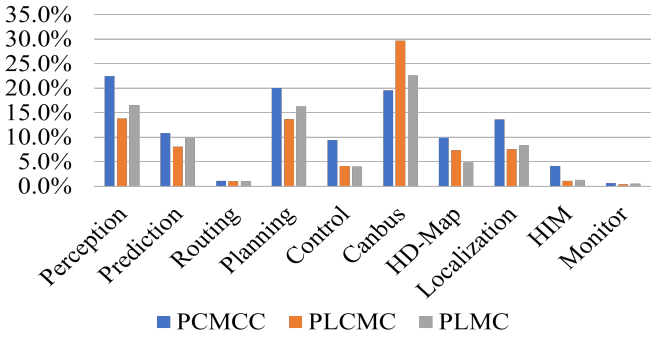

Fig. 8. PCMCC, PLCMC and PLMC of co-modified clones in Apollo

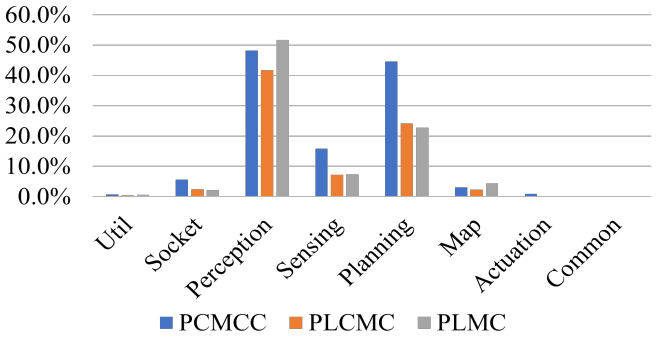

Fig. 9. PCMCC, PLCMC and PLMC of co-modified clones in Autoware

(Autoware) have a large number of code clones, which means that developers prefer to use code clones in these core modules in autonomous driving software. Meanwhile, clones in module *Perception, Planning, Canbus, and Sensing* are more likely to induce bugs and co-modifications, which indicates code clones in these modules of automated driving software deserve special attention.

RQ5: Do cross-module code clones exist in automated driving software? If so, whether cross-module clones are bug-prone and have co-modifications?

To reduce the complexity of auto-driving tasks and maintenance, an automated driving software is often decomposed into multiple modules, so that each module can complete its task and be maintained independently. If there are cross-module code clones (i.e. code fragments of clones spread over different modules), it means the involved modules are not really independent, they are implicitly coupled by cloned code. By examining the bug-proneness and co-modifications of cross-module clones, we explore whether code clones propagate bugs and modifications over different modules, which would undermine the Independence of modules' maintenance.

We have summarized the experimental results in Table VIII and IX. Column "NCMC" indicates the number of cross-module clones, and Column "PCMC" shows the proportion of cross-module clones in the version span. Column "PBCCM" shows the proportion of the bug-prone clones that are cross-module. Column "PCCCM" shows the proportion of co-modified clones that are cross-module. Taking Apollo as an example, we can see that 13.4% - 19.3% of code clones in this project are cross-module, meaning that to some extent code clones cause couplings among the modules. 13.6% to 22.5% of bug-prone clones are cross-module, indicating that the cross-module clones have brought bugs over different modules. Fixing such cross-module bugs requires locating and inspecting all the involved modules, which affects the efficiency of bug fixes. Moreover, 10.7% to 21.4% of co-modified clones are cross-module. That is, such cross-module cloned fragments need to be changed consistently in different modules. We can obtain similar observations for the Autoware project, with the ranges of PCMC, PBCCM, and PCCCM as 15.2% - 23.0%, 3.8% - 22.3%, and 10.9% to 29.0%, respectively.

Answer to RQ5: An average of 16.6% of the code clones in Apollo are cross-module clones, and the one of Autoware is 20.2%. It means code clones do cause couplings among modules in automated driving software. Meanwhile, an average of 19.5% (Apollo), 14.8% (Autoware) of bug-prone clones are cross-module, and an average of 15.7% (Apollo), 19.5% (Autoware) of co-modified clones are cross-module. These suggest that code clones could trigger cross-module bug fixes, and cause the demand for synchronous modifications to the code fragments in different modules, which would negatively affect the maintainability of automated driving software.

TABLE VIII
STATICS OF CROSS-MODULE CLONES IN APOLLO

VersionRange	*NCMC*	*PCMC*	*PBCCM*	*PCCCM*
1.0.0-3.0.0	39	13.4%	13.6%	10.7%
1.5.0-3.5.0	142	19.0%	21.1%	21.4%
2.0.0-5.0.0	268	19.1%	19.2%	16.5%
2.5.0-5.5.0	381	19.3%	21.4%	16.3%
3.0.0-6.0.0	371	17.9%	21.7%	18.9%
3.5.0-7.0.0	403	13.7%	22.5%	13.6%
5.0.0-8.0.0	505	14.0%	17.1%	12.3%
Average	301	16.6%	19.5%	15.7%

NCMC: the number of cross-module clones. *PCMC:* the proportion of cross-module clones. *PBCCM:* the proportion of bug-prone clones across different modules. *PCCCM:* the proportion of co-modified clones across different modules.

TABLE IX
STATICS OF CROSS-MODULE CLONES IN AUTOWARE

VersionRange	*NCMC*	*PCMC*	*PBCCM*	*PCCCM*
1.0.0-1.4.0	206	15.2%	3.8%	10.9%
1.1.0-1.5.0	248	16.8%	6.8%	14.8%
1.2.0-1.6.0	424	20.0%	14.6%	17.4%
1.3.0-1.7.0	493	21.3%	21.4%	22.1%
1.4.0-1.8.0	497	21.5%	13.8%	22.1%
1.5.0-1.9.0	506	21.5%	14.2%	20.6%
1.6.0-1.10.0	555	22.4%	21.5%	19.2%
1.7.0-1.11.0	609	23.0%	22.3%	29.0%
Average	442	20.2%	14.8%	19.5%

V. Discussion

A. Findings

Based on the previous findings, we discuss the significance of our experimental results. In particular, we will draw lessons from our findings that may guide future work related to code clone analysis of automated driving software.

Finding 1 illuminates the prevalence of code clones in automated driving software. In conclusion, code clones are commonly happened in both studied projects, about an average of 30% of code fragments are involved in code clones, and more than 50% of files contain code clones. This indicates that, due to the special development characteristics of automated driving software, such as the manipulation of data from multiple sensors or hardware devices, the software has a high potential of getting code cloned. Meanwhile, the number of code clones and the proportion of cloned lines showed an upward trend for different versions, meaning that code clones have continuously existed as software evolves.

Finding 2 presents the bug-proneness of code clones in automated driving software. 5.4%-55.9% of code clones in Apollo and 5.9%-18.7% of code clones in Autoware have been involved in bugs, and the average fixing time of clone-related bugs is much larger than that of bugs not related to clones. It suggests that code clones have contributed to the bug-proneness of automated driving software, and code cloning increases the time for fixing bugs, thus negatively affecting the maintainability of the software system. If researchers can propose effective cloned code reviews and risk assessment standards to capture bug-prone code clones, the bugs related to clones can be monitored and repaired in a timely manner.

Finding 3 demonstrates the existence of co-modified code clones in the automated driving software. 20.2%-77.2% of the code clones in Apollo and 8.2%-13.8% of the code clones in Autoware have caused co-modifications. This indicates that developers need to spend time dealing with consistency and compilation requests. To maintain the correctness of the software, the cloned fragments should be consistently tracked and modified to avoid problems caused by inconsistent changes. If software engineering research can effectively manage code clones and effectively refactor them, the problems caused by inconsistency could be reduced.

Finding 4 presents how code clones distribute in the modules of automated driving software. The code clones in Apollo mainly concentrate on *Perception*, *Planning*, and *Canbus* modules, accounting for 26.2%, 20.4%, and 21.2% respectively. While the most involved modules of code clones in Autoware are *Perception*, *Planning*, and *Sensing* modules, with the percentage of 53.8%, 36.7%, and 17.9%, respectively. Moreover, these modules contain more bug-prone and co-modified code clones than others. It suggests that in the automated driving software, code clones have a greater impact on the maintainability of modules related to Canbus, perception, planning, and sensing tasks. Developers need to pay special attention to code clones in these modules, and monitor the risk of code cloning, to ensure the normal operation of important functional modules, thereby promoting the maintenance and security of the automated driving software.

Finding 5 presents the cross-module code cloning in automated driving software. In Apollo, 16.6% of the code clones on average in are cross-module clones, 19.5% of bug-prone clones and 15.7% of co-modified clones are cross-module. In Autoware, the percentages are 20.2%, 14.8%, and 19.5%, respectively. This suggests that code cloning increases the coupling between modules. Meanwhile, the existence of code clones has made changes propagated among different modules, and even introduced bugs into different modules. Therefore, there is a demand to well manage and refactor cross-module clones, so that developers can alleviate the impacts of couplings caused by code clones, especially reduce the co-modifications and bugs across modules, and prevent unexpected interference among modules.

B. Future Work

According to our experimental approaches and findings, we discuss the possible analyses to facilitate future improvements and research. First, a future study can be conducted on the trade-offs of clones' usage during development. That is, we can conduct interviews with developers in the automated driving software to know the trade-offs in applying code cloning during development, which in turn can help practitioners adopt safer code reuse practices.

Second, our experimental results provide the number of clones, the proportion of clones, and the proportion of bug-prone clones along with the project versions, which can be used for analyzing the evolution mode of code clones in future work. Meanwhile, since the prevalence of clones varies

over different releases, it will be interesting to examine which code clones have been removed or added, and what kind of strategies were used in modifying cloned code fragments.

Third, our study examined the bug-fixing commits related to code clones (commit id, average fixing time, message, involved modules, patch code), through the analysis of commit information, we believe that it can provide insights into the relationship between code clones and bugs. It is useful for researchers to assess the possibility of code clones in bug fixes. In addition, a deeper study can be conducted on the relationship between code clones and bugs in automated driving software. For example, why the cloned code fragments are involved in bug fixes, whether there are fixing patterns on the code clones, or how the code clones propagate bug-proneness in automated driving systems. Combined with fixing patterns mined from the bug-fixing commit, we can provide guidance for developers in bug testing, fixing, code refactoring, and clone management in the automated driving system.

Finally, we take the modular structure of automated driving systems into account. In the future, we can analyze why the code clones mainly locate in a few modules, the root causes of cross-module clones, and how to remove such cross-module clones. Meanwhile, these analyses can be extended to other research fields. For example, which modules have more bugs, whether and to what extent the architecture design impacts bugs' occurrences, and whether there exist architectural smells special to automated driving software.

VI. Threats to Validity

First, our work leverages Simian [28] to detect code clones in the projects. The configuration settings of this tool could affect its detection results, which in turn causes an internal threat. In this paper, We set the configurations by following the work of Ragkhitwetsagul et al. [34], which can achieve a high recall and precision.

Second, because the main source codes of the selected systems are both written in C++, our clone detection objects are mainly C++ source files and header files. So we do not guarantee that our experimental results are applicable to systems written in other languages. But most of the current automated driving software systems are developed based on ROS, and developers are more likely to use C++ for development.

Third, an external validity is from our dataset. We only investigated two software systems, Apollo and Autoware. But they are the two most widely-used and representative automated driving systems by far. Autoware has been currently used by more than 100 companies and applied on more than 30 vehicles in more than 20 different countries around the world [40]. More than 220 global partners have used the Apollo services [41]. In addition, both Apollo and Autoware have a modular architecture and an extensive and publicly accessible repository [19], [20]. So that we can study sufficient data containing dozens of releases with 640,000 LOC and more than 23,000 commits. Thus, we have confidence that our findings are likely to be representative and generalizable to other automated driving software systems.

VII. Related Work

A. Consistency of Code clone

Our work explores the co-modifications in code clones, we were interested in reviewing the literature on aspects of maintaining clone consistency in software development.

Mo et al. [42] studied the existence of code cloning and co-modification within and across services in microservice systems, providing a basic understanding of the condition of code cloning in the micro-service systems.

Zhang et al. [37] proposed a definition of clone consistency maintenance requirements and provided a modified set of attributes using machine learning methods to improve the predictive power of clone consistency aspects.

Hu et al. [7] conducted an exploratory study on clone change propagation of five well-known open source projects, and proposed a graph-based deep learning method to predict the change propagation needs of cloned instances to accurately and efficiently make changes between cloned instances.

Alexander et al. [43] conducted an empirical study to quantify the potential for automating variant synchronization in cloning and possession. And quantified potential for automating variant synchronization in cloning and possession.

The above studies explore the consistent changes in code clones, and present the importance of maintaining consistency in cloned code. In our work, we extend such analysis to automated driving software for investigating the existence and percentages of co-modifications in code clones.

B. Code clones Analysis

Inconsistent changes may cause bugs in the clone, or there may be cases where bugs are propagated from the original fragment to the cloned fragment. Code clones can affect software development, quality, and maintenance. Therefore, there are many studies exploring the bug-proneness of code clones.

Jebnoun et al. [10] analyzed the existence and distribution of clones in machine learning systems, studying the correlation between bugs and code clones to evaluate the impact of clones on the quality of the machine learning systems.

Islam et al. [44] detect the genealogy and change patterns of code clones to extract the evolutionary characteristics of code clones, and automatically analyze their evolutionary characteristics to pinpoint defects in the software system code base code clone.

Ye et al. [45] took the clone code of continuous multi-version software as the analysis object, and proposed a method of building a "clone code case library" based on bugs, which is helpful for more effective reuse, refactoring, and evolution management of clone code.

Additionally, there are a few studies on code clone detection in IoT systems that are relevant to the automated driving domain. Zhang et al. [46] presented that, due to the large scale and variety of IoT devices, developers prefer to use

code clones to quickly complete tasks. Cloning could be a serious threat in the Internet of Things (IoT), since an attacker can simply gather configuration and authentication credentials from a non-tamper-proof node, and replicate it in the network [47].

Tekchandani et al. [48] proposed a novel approach that finds semantic code clones in IoT applications. Luo et al. [49] proposed and implement a prototype, GeneDiff, which is a semantic-based representation binary clone detection approach for cross-architectures in IoT.

Ullah et al. [50] proposed a hybrid approach to the Control Flow Graph (CFG) and a deep-learning model to secure the smart services of the 5G-IoT framework.

From the above studies, we can find that code clones can propagate bugs, vulnerability to different locations via the cloned fragments. Similarly, we also analyzed the bug-prone code clones in automated driving software to explore whether and to what extent the bugs are propagated over code clones.

C. Automated driving software quality

The continuous development of autonomous driving has aggravated the complexity of the internal software system of a vehicle. The direct consequence of this software complexity is that there may be defects in software design or implementation. Consequently, there are more and more studies on the quality of automated driving software.

Zhou Y et al. [51] proposed the AV Unit framework, with a modular design to support different simulators, for systematically testing autonomous driving systems according to customizable correctness specifications.

Lou G et al. [52] classified the existing autonomous driving system testing techniques, showed 7 common practices and 4 emerging requirements for autonomous driving system, and analyzed the gap between autonomous driving system research and practitioners' needs.

Gan Y et al. [53] conducted a comprehensive fault injection to study the behavior of the autonomous driving software stack under different error sources, proposing a dynamic protection system that devotes a limited protection budget to the most vulnerable parts of the autonomous driving software at a high level, thereby reducing the error rate of the autonomous driving software stack.

Unlike the above studies, which mainly analyzed the software quality of the automated driving software from the perspective of bug classification, testing tools, or vulnerable parts. Our work focuses on investigating the impact of code clones in automated driving software. We examine the existence, bug-proneness, and co-modifications of code clones. Meanwhile, considering the modular structure adopted by automated driving software, we further examine how code clones distribute over different modules and the cross-module clones.

VIII. Conclusion

In this paper, we detect and analyze code clones in automated driving software by examining their existence, bug-proneness, and co-modifications. In addition, we take the software modules into account by analyzing the code clones within or across different modules.

Through the clone analysis of *Apollo* and *Autoware*, we have found that: 1) code clones commonly happen in automated driving software. about 30% code lines are cloned, and more than 50% of files contain code clones; 2) a portion of code clones in the automated driving software have caused bugs, and the clone-related bugs require more time to get fixed than the bugs not related to clones; 3) about 36.4% of clones in Apollo and 11.2% of clones in Autoware have caused co-modifications, meaning there is a demand to consistently track and manage such clones in automated driving software. 4) When considering each module individually, we have found the *Perception*, *Planning Canbus* and *Sensing* modules are more likely to encounter code clones, and code clones in these modules are more likely to induce bugs and co-modifications. Therefore, developers need to pay more attention to managing and maintaining clones in these modules. 5) even if the automated driving software systems have been designed to be modular, there still exist cross-module code clones, which cause couplings between modules. Meanwhile, they have propagated bugs and co-modifications among different modules, which violates the modularity.

In conclusion, although code clones may lead to productivity gains, developers should be aware of the problems caused by them. Based on our analysis of the existence, bug-proneness, and co-modifications of code clones, we believe that our work has provided informative insights into code clones of autonomous driving software.

Acknowledgments

This work is supported by the National Natural Science Foundation of China under grant No. 62002129, the Knowledge Innovation Program of Wuhan-Shuguang Project under grant No. 2022010801020280, and the Natural Science Foundation of Hubei Province of China under grant No. 2021CFB577.

References

[1] A. Tampuu, T. Matiisen, M. Semikin, D. Fishman, and N. Muhammad, "A survey of end-to-end driving: Architectures and training methods," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 4, pp. 1364–1384, 2020.

[2] E. Yurtsever, J. Lambert, A. Carballo, and K. Takeda, "A survey of autonomous driving: Common practices and emerging technologies," *IEEE access*, vol. 8, pp. 58 443–58 469, 2020.

[3] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

[4] M. Kondo, G. A. Oliva, Z. M. Jiang, A. E. Hassan, and O. Mizuno, "Code cloning in smart contracts: a case study on verified contracts from the ethereum blockchain platform," *Empirical Software Engineering*, vol. 25, pp. 4617–4675, 2020.

[5] M. Hammad, Ö. Babur, H. A. Basit, and M. Van Den Brand, "Clone-seeker: Effective code clone search using annotations," *IEEE Access*, vol. 10, pp. 11 696–11 713, 2022.

[6] R. Tairas and J. Gray, "An information retrieval process to aid in the analysis of code clones," *Empirical Software Engineering*, vol. 14, pp. 33–56, 2009.

[7] B. Hu, Y. Wu, X. Peng, C. Sha, X. Wang, B. Fu, and W. Zhao, "Predicting change propagation between code clone instances by graph-based deep learning," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 2022, pp. 425–436.

[8] G. Shobha, A. Rana, V. Kansal, and S. Tanwar, "Comparison between code clone detection and model clone detection," in *2021 9th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO)*. IEEE, 2021, pp. 1–5.

[9] V. Bandi, C. K. Roy, and C. Gutwin, "Clone swarm: A cloud based code-clone analysis tool," in *2020 IEEE 14th International Workshop on Software Clones (IWSC)*. IEEE, 2020, pp. 52–56.

[10] H. Jebnoun, M. S. Rahman, F. Khomh, and B. A. Muse, "Clones in deep learning code: what, where, and why?" *Empirical Software Engineering*, vol. 27, no. 4, p. 84, 2022.

[11] M. Mondal, C. K. Roy, B. Roy, and K. A. Schneider, "Fleccs: A technique for suggesting fragment-level similar co-change candidates," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 160–171.

[12] F. Zhang and S.-C. Khoo, "An empirical study on clone consistency prediction based on machine learning," *Information and Software Technology*, vol. 136, p. 106573, 2021.

[13] Y. Xu, T. Zhang, and Y. Bao, "Analysis and mitigation of function interaction risks in robot apps," in *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, 2021, pp. 1–16.

[14] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi, "Autoware on board: Enabling autonomous vehicles with embedded systems," in *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 2018, pp. 287–296.

[15] A. Santos, A. Cunha, N. Macedo, S. Melo, and R. Pereira, "Variability analysis for robot operating system applications," in *2022 Sixth IEEE International Conference on Robotic Computing (IRC)*. IEEE, 2022, pp. 111–118.

[16] P. Estefó, R. Robbes, and J. Fabry, "Code duplication in ros launchfiles," in *2015 34th International Conference of the Chilean Computer Science Society (SCCC)*. IEEE, 2015, pp. 1–6.

[17] Y. Su and L. Wang, "Integrated framework for test and evaluation of autonomous vehicles," *Journal of Shanghai Jiaotong University (Science)*, vol. 26, pp. 699–712, 2021.

[18] G. M. Selim, L. Barbour, W. Shang, B. Adams, A. E. Hassan, and Y. Zou, "Studying the impact of clones on software defects," in *2010 17th Working Conference on Reverse Engineering*. IEEE, 2010, pp. 13–21.

[19] Baidu, "Apolloauto," https://github.com/ApolloAuto/apollo Accessed April 13, 2023.

[20] TheAutowareFoundation, "Autoware," https://github.com/autowarefoundation/autoware Accessed April 13, 2023.

[21] C. Gao, G. Wang, W. Shi, Z. Wang, and Y. Chen, "Autonomous driving security: State of the art and challenges," *IEEE Internet of Things Journal*, vol. 9, no. 10, pp. 7572–7595, 2021.

[22] G. Shobha, A. Rana, V. Kansal, and S. Tanwar, "Code clone detection—a systematic review," *Emerging Technologies in Data Mining and Information Security: Proceedings of IEMIS 2020, Volume 2*, pp. 645–655, 2021.

[23] Baidu, "Canbus: fixed bug 6(1) from safety team." https://github.com/ApolloAuto/apollo/commit/2e76d6cce2df8ba06d7ee0c04f1b4237f6b8be22 Accessed April 13, 2023.

[24] Y. Zhong, X. Zhang, W. Tao, and Y. Zhang, "A systematic literature review of clone evolution," in *Proceedings of the 5th International Conference on Computer Science and Software Engineering*, 2022, pp. 461–473.

[25] M. Nadim, M. Mondal, C. K. Roy, and K. A. Schneider, "Evaluating the performance of clone detection tools in detecting cloned co-change candidates," *Journal of Systems and Software*, vol. 187, p. 111229, 2022.

[26] B. Hu, Y. Wu, X. Peng, J. Sun, N. Zhan, and J. Wu, "Assessing code clone harmfulness: Indicators, factors, and counter measures," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 225–236.

[27] M. Alcon, H. Tabani, J. Abella, and F. J. Cazorla, "Enabling unit testing of already-integrated ai software systems: The case of apollo for autonomous driving," in *2021 24th Euromicro Conference on Digital System Design (DSD)*. IEEE, 2021, pp. 426–433.

[28] S. Harris, "Simian – similarity analyzer, version 2.5.10," http://www.harukizaemon.com/simian/index.html,2018 Accessed Oct 1, 2022.

[29] C. Ragkhitwetsagul, J. Krinke, and B. Marnette, "A picture is worth a thousand words: Code clone detection based on image similarity," in *2018 IEEE 12th International workshop on software clones (IWSC)*. IEEE, 2018, pp. 44–50.

[30] J. Krinke and C. Ragkhitwetsagul, "Code similarity in clone detection," *Code Clone Analysis: Research, Tools, and Practices*, pp. 135–150, 2021.

[31] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool, "A systematic review on code clone detection," *IEEE access*, vol. 7, pp. 86 121–86 144, 2019.

[32] M. Ciniselli, L. Pascarella, and G. Bavota, "To what extent do deep learning-based code recommenders generate predictions by cloning code from the training set?" in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 167–178.

[33] Q. Hum, W. J. Tan, S. Y. Tey, L. Lenus, I. Homoliak, Y. Lin, and J. Sun, "Coinwatch: A clone-based approach for detecting vulnerabilities in cryptocurrencies," in *2020 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 2020, pp. 17–25.

[34] C. Ragkhitwetsagul and J. Krinke, "Siamese: scalable and incremental code clone search via multiple code representations," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2236–2284, 2019.

[35] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: analytics and risk prediction of software commits," in *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, 2015, pp. 966–969.

[36] D. Spadini, M. Aniche, and A. Bacchelli, "Pydriller: Python framework for mining software repositories," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 908–911.

[37] F. Zhang, S.-C. Khoo, and X. Su, "Improving maintenance-consistency prediction during code clone creation," *IEEE Access*, vol. 8, pp. 82 085–82 099, 2020.

[38] Baidu, "Apollo-document," https://apollo.baidu.com/community/Apollo-Homepage-Document/Apollo_Doc_CN_8_0 Accessed Apr 13, 2023.

[39] TheAutowareFoundation, ""autowarefoundation/autoware at 1.11.0," github," https://github.com/autowarefoundation/autoware Accessed Apr 13, 2023.

[40] ——, "Autoware overview," https://autoware.org/autoware-overview Accessed April 13, 2023.

[41] Baidu, "Apolloautoaboutus," https://www.apollo.auto/aboutus Accessed April 13, 2023.

[42] R. Mo, Y. Zhao, Q. Feng, and Z. Li, "The existence and co-modifications of code clones within or across microservices," in *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2021, pp. 1–11.

[43] A. Schultheiß, P. M. Bittner, T. Thüm, and T. Kehrer, "Quantifying the potential to automate the synchronization of variants in clone-and-own," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2022, pp. 269–280.

[44] M. H. Islam, R. Paul, and M. Mondal, "Predicting buggy code clones through machine learning," in *Proceedings of the 32nd Annual International Conference on Computer Science and Software Engineering*, 2022, pp. 130–139.

[45] Z. Ye, Z. Yuwu, and L. D. Sheng, "A method for constructing a clone code case library based on bugs," in *2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*. IEEE, 2019, pp. 2480–2486.

[46] H. Zhang and K. Sakurai, "A survey of software clone detection from security perspective," *IEEE Access*, vol. 9, pp. 48 157–48 173, 2021.

[47] P.-Y. Lee, C.-M. Yu, T. Dargahi, M. Conti, and G. Bianchi, "Mdsclone: multidimensional scaling aided clone detection in internet of things," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 8, pp. 2031–2046, 2018.

[48] R. Tekchandani, R. Bhatia, and M. Singh, "Semantic code clone detection for internet of things applications using reaching definition and liveness analysis," *The Journal of Supercomputing*, vol. 74, pp. 4199–4226, 2018.

[49] Z. Luo, B. Wang, Y. Tang, and W. Xie, "Semantic-based representation binary clone detection for cross-architectures in the internet of things," *Applied Sciences*, vol. 9, no. 16, p. 3283, 2019.

[50] F. Ullah, M. R. Naeem, L. Mostarda, and S. A. Shah, "Clone detection in 5g-enabled social iot system using graph semantics and deep learning model," *International Journal of Machine Learning and Cybernetics*, vol. 12, pp. 3115–3127, 2021.

[51] Y. Zhou, Y. Sun, Y. Tang, Y. Chen, J. Sun, C. M. Poskitt, Y. Liu, and Z. Yang, "Specification-based autonomous driving system testing," *IEEE Transactions on Software Engineering*, 2023.

[52] G. Lou, Y. Deng, X. Zheng, M. Zhang, and T. Zhang, "Testing of autonomous driving systems: where are we and where should we go?" in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 31–43.

[53] Y. Gan, P. Whatmough, J. Leng, B. Yu, S. Liu, and Y. Zhu, "Braum: Analyzing and protecting autonomous machine software stack," in *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2022, pp. 85–96.