Compiler Test-Program Generation via Memoized Configuration Search

Junjie Chen
College of Intelligence and Computing
Tianjin University
Tianjin, China
junjiechen@tju.edu.cn

Chenyao Suo
College of Intelligence and Computing
Tianjin University
Tianjin, China
chenyaosuo@tju.edu.cn

Jiajun Jiang†
College of Intelligence and Computing
Tianjin University
Tianjin, China
jiangjiajun@tju.edu.cn

Peiqi Chen
College of Intelligence and Computing
Tianjin University
Tianjin, China
chenpeiqi@tju.edu.cn

Xingjian Li
College of Intelligence and Computing
Tianjin University
Tianjin, China
xingjianli@tju.edu.cn

Abstract*—To ensure compilers' quality, compiler testing has received more and more attention, and test-program generation is the core task. In recent years, some approaches have been proposed to explore test configurations for generating more effective test programs, but they either are restricted by historical bugs or suffer from the cost-effectiveness issue. Here, we propose a novel test-program generation approach (called MCS) to further improving the performance of compiler testing. MCS conducts memoized search via multi-agent reinforcement learning (RL) for guiding the construction of effective test configurations based on the memoization for the explored test configurations during the on-the-fly compiler-testing process. During the process, the elaborate coordination among configuration options can be also well learned by multi-agent RL, which is required for generating bug-triggering test programs. Specifically, MCS considers the diversity among test configurations to efficiently explore the input space and the testing results under each explored configuration to learn which portions of space are more bug-triggering. Our extensive experiments on GCC and LLVM demonstrate the performance of MCS, significantly outperforming the state-of-the-art test-program generation approaches in bug detection. Also, MCS detects 16 new bugs on the latest trunk revisions of GCC and LLVM, and all of them have been confirmed or fixed by developers. MCS has been deployed by a global IT company (i.e., *Huawei*) for testing their in-house compiler, and detects 10 new bugs (covering all the 5 bugs detected by the compared approaches), all of which have been confirmed.

***Index Terms*—Compiler Testing, Test Program Generation, Reinforcement Learning, Configuration**

I. Introduction

Compilers are one of the most fundamental software, because almost all the software are required to be processed by compilers before using them. However, like other software [1], [2], compilers also contain bugs and meanwhile compiler bugs are very harmful due to their fundamental role [3]. On one hand, compiler bugs could cause any software built on them to produce unexpected behaviors; on the other hand, compiler bugs could aggravate the debugging difficulty as it is hard for developers to distinguish whether a software failure is caused by the software they are developing or the compiler they are using. Hence, it is important to ensure the quality of compilers.

Compiler testing is the most widely-used way of ensuring compilers' quality and many compiler testing techniques have been proposed [3]–[10], in which automated test-program generation is the core task. Over the years, some test-program generators have been developed (e.g., Csmith [11] and CLsmith [12]). They generate a large number of test programs by depending on a test configuration to control what program features are likely to be included. A test configuration consists of a set of options, each of which controls how likely a program feature can be included in a generated test program. For example, a test configuration of Csmith [11] (one of the most widely-used test-program generators) consists of 71 options, reflecting the probabilities of these program features (e.g., the `goto` statement) to be included.

In general, a test-program generator provides a default test configuration, which is determined by developers according to their experience, for the practical use [11], [13], [14]. However, as the input space is enormous (including all syntactically correct programs), relying on only one configuration is scarcely possible to sufficiently explore the entire input space within a limited testing period, causing that only a limited number of compiler bugs can be detected [13], [14]. To relieve this limitation, some approaches have been proposed to explore test configurations so that as many compiler bugs can be detected as possible [13]–[16]. However, they still suffer from the effectiveness issue. Besides the enormous input space, another reason is that the triggering of a compiler bug tends to involve the specific combination of some program features [17], indicating that the options in a test configuration require elaborate coordination for generating bug-triggering test programs. Specifically, some approach randomly constructs a test configuration before generating a test program during the online testing process [14], but random search can

†Corresponding author.

hardly capture useful information for elaborate coordination among options and compiler bugs are not evenly distributed across input space [13], [18], leading to less effective. More advanced approaches infer a set of test configurations by mining historical bugs in an offline way [13], [16], but their effectiveness is limited by the mined history information, i.e., missing to detect unseen bugs due to lack of consideration of the characteristics of the current version under test. Hence, faced with the goal of detecting as many compiler bugs as possible, more effective approaches are desirable.

In this work, we propose a novel test-program generation approach, called **MCS** (**M**emoized **C**onfiguration **S**earch). It aims to improve the performance of compiler testing by incorporating the explored test configurations during the on-the-fly compiler-testing process for guiding the construction of the next configuration and intelligently capturing elaborate coordination among options. Specifically, instead of *offline* inferring a set of test configurations by mining historical bugs, MCS interleaves the process of searching for test configurations and the online testing process. To more efficiently explore the input space, MCS considers the diversity between the next configuration and the explored ones. Also, it considers the testing results (i.e., triggering bugs or not) under each explored configuration to learn which portions of space are more bug-triggering, and then can pay more attention to exploiting those portions of space. That is, MCS conducts *memoized search* to construct the next *diverse and bug-triggering* test configuration based on the memoization for the explored ones along with the on-the-fly testing process.

To continuously incorporate the knowledge from explored test configurations and effectively capture coordination among options, MCS innovatively adopts the *multi-agent* reinforcement learning (RL) method (i.e., multi-agent A2C [19]) to achieve our search goal. Indeed, the existing study has demonstrated that RL is effective for memoized search [20] and multi-agent RL can effectively seize mutual effects among options by treating each option as an individual agent. Under the framework of RL, MCS incorporates both diversity among configurations and bug-triggering results as the reward to measure the quality of each configuration. Also, MCS empirically investigates options' negative effect on compiler testing via a preliminary study and also considers it in the reward function, so as to avoid generating test programs that negatively affect compiler testing. Overall, MCS could make each constructed configuration as effective as possible by gradually learning from the memoization for the explored configurations along with the on-the-fly testing process, so that a wider range of bugs can be detected within the given testing period.

To evaluate the performance of MCS, we conducted an extensive study on two popular C compilers (i.e., GCC and LLVM) and the most widely-used test-program generator (i.e., Csmith [11]) following the existing studies [11], [17], [21]–[23]. Our results show that MCS significantly outperforms two state-of-the-art compiler test-program generation approaches, i.e., HiCOND (that offline infers a set of test configurations based on historical bugs) and Swarm Testing (that randomly constructs test configurations for test-program generation during the online testing process). For example, MCS detects 105.56% and 311.11% more bugs than them respectively, and 59.46% of bugs detected by MCS are unique. We also applied MCS to test the latest revisions of GCC and LLVM, and then it detected 16 new bugs during three-month testing. After reporting them, all of these bugs have been confirmed/fixed by developers. In particular, MCS has been deployed by a global IT company (i.e., *Huawei*) to test their in-house compiler, which is a high-performance and easy-to-expand compiler for general-purpose processor architectures (we hide the compiler name due to the company policy). During 10-day testing, MCS detects 10 new bugs and all of them have been confirmed by developers, but the state-of-the-art compared approaches just detect at most 5 bugs and all of them are covered by MCS. The results further confirm the practical value of MCS.

Our work makes the following major contributions:

- We propose the first RL-based test-program generation approach, which conducts an online memoized search for bug-triggering and diverse test configurations by learning from explored ones during the on-the-fly testing process.
- We conducted an extensive study on GCC and LLVM, demonstrating the superiority of MCS over the state-of-the-art compared approaches. In particular, MCS detected 16 new bugs on the latest revisions of GCC and LLVM, all of which have been confirmed/fixed by developers.
- We have deployed MCS to a global IT company (i.e., *Huawei*) for testing their in-house high-performance compiler. To further promote its practical use and future research, we have released our tool at **https://github.com/tju-chenyaosuo/MCS**.

II. Background

A. Test Programs and Test Configurations

Test-program generation is the initial step in various compiler testing techniques. After generating a test program, various test oracle mechanisms (e.g., differential testing [24] or metamorphic testing [25]) can be adopted to determine whether it triggers a compiler bug. That is, the generated test programs could largely affect the overall performance of compiler testing. Due to its important role, plenty of efforts have been devoted to developing various test-program generators, such as Csmith [11], CLsmith [12], and DeepSmith [26].

Different from the test inputs of other software (e.g., numbers or strings), test programs involve various program features (e.g., different kinds of statements) [11]. Typically, test-program generators depend on test configurations to generate a large number of test programs [13]–[15]. A test configuration controls what program features are likely to be included in a generated test program. A test configuration consists of a set of options and each option reflects the probability of the program feature to be included. For example, one of the most widely-used test-program generators, i.e., Csmith [11], provides a test configuration containing 71 options to control the generation of test programs, e.g., the probability determining whether a type (e.g., `int` or `struct` types) is generated.

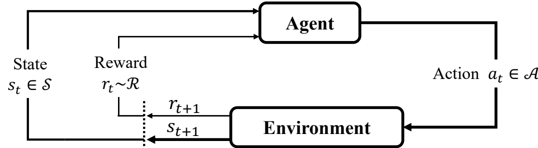


Fig. 1. Interaction between agent and environment.

Instead of the default test configuration provided by the developers of a test-program generator, some approaches have been proposed to explore test configurations so as to explore the input space more effectively, which can be divided into two categories:

- *Offline approach*, produces a set of test configurations in an offline way, instead of the default one, for compiler testing. They utilize historical bugs to infer a set of test configurations [13], [16].
- *Online approach*, constructs test configurations during online testing. The current approach randomly constructs a configuration when generating a test program [14]. That is, it conducts random search in the input space.

As presented in Section I, they still suffer from the effectiveness issue. Please note that our approach belongs to the second category by conducting memoized search for the next diverse and bug-triggering configuration based on the memoization for the explored ones during the on-the-fly testing process. It can help explore the input space more efficiently.

B. Reinforcement Learning

As MCS adopts multi-agent RL to guide the search process, in this section we first introduce some background of RL, and then explain why we choose RL (i.e., multi-agent RL).

Background of RL. RL aims to learn an optimal control policy for agents that interact with an unknown environment. The target is to choose a sequence of actions that maximize the cumulative reward for agents in a long run [27]–[29]. Formally, this process is represented as a Markov Decision Process (shown in Figure 1) that is defined by a tuple ($\mathcal{S}$, $\mathcal{A}$, $\mathcal{P}$, $\mathcal{R}$, γ), where $\mathcal{S}$ is a set of states that an agent can attain in a specific environment, while $\mathcal{A}$ is a set of actions the agent can perform. During the RL process, at each time step t, an agent takes an action $a_t \in\mathcal{A}$ based on the observation of a state $s_t \in\mathcal{S}$ and moves to the next state s_{t+1} with a *transition probability* of $\mathcal{P}^a_{ss'}$: $\mathcal{P}^a_{ss'} = P_r\{s_{t+1} = s'|s_t = s, a_t = a\}$. Then, a feedback reward r_t for the taken action will be received via a predefined reward function $\mathcal{R}(s_t, a_t, s_{t+1})$. The above process proceeds until the agent reaches the terminal, e.g., exceeding a given time budget. Suppose that the duration of an episode is from time t to T, and the cumulative reward obtained by the agent in this episode can be obtained by: R_t=$\Sigma^T_{t'=t}\gamma^{t'-t}r_t$ where γ is a discount factor that determines the importance of the current and future rewards, ranging $[0, 1]$.

Why choose RL? A typical search scenario is to first adopt some search algorithm to find the optimal solution with regard to the defined fitness function, and then apply the solution to obtain the optimal result, e.g., (search-based) compiler autotuning [30]–[34]. That is, the typical scenario consists of two stages: solution search and solution use, which have a clear *precedence* relationship.

Different from the typical search scenario, our task focuses on on-the-fly compiler testing and thus *interleaves* the solution search process (i.e., constructing the next test configuration based on the memoization for the explored ones) and the solution use process (i.e., applying the test configuration to generate test programs for compiler testing). That is, during the search process, when generating a test configuration, its quality is immediately measured by using it to generate test programs for compiler testing and then it will be memoized for guiding follow-up configuration construction. When the search process terminates, the on-the-fly testing process also terminates and then the detected bugs during the process are outputted. Hence, to explore the input space more efficiently for better compiler testing, we should ensure that each constructed test configuration during the search process is as effective as possible in our scenario.

Please note that our task does not match the above typical search scenario. Under the typical scenario, it has to await the costly search process before starting the on-the-fly compiler-testing process. It can delay the testing process and thus negatively affect the performance of compiler testing [22], compared with our scenario of interleaving both processes.

To complete our task, we adopt RL as (1) it is suitable to our scenario by taking test configurations as states, designing the ways of updating option settings as actions, and considering the quality of explored configurations in the reward function. In this way, RL can help construct the next effective test configuration by predicting actions in the current state based on the memoization for explored ones during the on-the-fly testing process; (2) it has been demonstrated to be effective for memoized search [20]. In particular, we conducted an experiment to confirm the contribution of RL in MCS by comparing it with conventional search algorithms (Section VII-A). Please note that there is a slight difference between the original purpose of RL and our purpose for using RL. The former is to learn the control policy, while the latter is to construct as effective a test configuration as possible at each time step by using the gradually-learned policy.

We employ the A2C algorithm [19] in MCS as it is effective by combining the strengths of both value-based RL algorithms [35] and policy-based RL algorithms [36], and is suitable for our task under a single-thread running environment. A2C has a separate memory structure to explicitly represent the policy independent of the value function and incorporates knowledge from all possible actions to reduce variances of neural networks. The policy structure is known as the *actor*, which is used to select actions via an actor neural network (ANN), and the estimated value function is known as the *critic*, which criticizes the actions made by the *actor* via a critic neural network (CNN).

Why choose multi-agent RL? As presented by the existing work [17], the triggering of a compiler bug tends to involve the specific combination of several program features, and thus

elaborate coordination among options in a test configuration is required for effectively generating test programs. Hence, we adopt the multi-agent framework of A2C to seize mutual effects among options by treating each option as an individual agent, which maintains independent ANN and CNN but shares the same *environment* for coordination. Specifically, when given an initial test configuration (i.e., s_t), the learned multi-agent RL model can generate a set of actions to update the option settings respectively and produce a new configuration (i.e., s_{t+1}), which is expected to be more effective for test-program generation. Although the coordination in multi-agent RL could incur extra cost, the time cost is significantly less than that spent on compiling and executing test programs. Therefore, it may not shadow the ultimate optimization goal obviously. Besides, as presented by the existing work [37], multi-agent RL outperforms single-agent RL especially when the state-action space of single-agent RL is large (like our scenario), as the large state-action space can cause both ANN and CNN hard-to-converge, leading to inaccurate prediction. In the future, we can empirically compare multi-agent RL with single-agent RL in our scenario.

Although the theory of multi-agent A2C is mature, it is non-trivial to formulate our task under the RL framework and design an effective reward function to address our task well. Also, we are the first to address the task of exploring test configurations for better compiler testing based on *guided search* in an *online* process, showing the novelty of our work.

III. Approach

Although existing test-program generation approaches have made attempts to explore test configurations to improve compiler testing performance, they either are limited by historical bugs or suffer from the cost-effectiveness issue due to aimless search. Also, it is hard for them to effectively capture elaborate coordination among options for generating bug-triggering test programs. To achieve better compiler testing, we propose a novel test-program generation approach, called **MCS**, to conducting memoized search for effective test configurations based on the memoization for the explored ones during the on-the-fly compiler-testing process. Specifically, it adopts multi-agent RL for memoized configuration search, which also intelligently seizes mutual effects among options, so as to more efficiently explore the input space (especially the space involving bug-triggering test programs). It considers both diversity among configurations and testing results under each explored configuration as the reward to measure the quality of each configuration, thus guiding the construction of better test configurations in the follow-up process. Note that MCS does not use test coverage as the guidance since collecting coverage can incur more extra costs and the existing work has demonstrated that the test programs detecting more bugs do not improve line, branch, function coverage significantly [11].

In addition, there are a number of options in a test configuration and an option has a large value range. The huge configuration space and complex mutual effects among options can promote the generation of bug-triggering test programs, but it may also lead to the generation of negative-effect test programs for compiler testing. For example, it may generate the test program running for an extremely long time, which has negative effect on the performance of compiler testing as demonstrated in the existing study [22]. Hence, we carefully investigate options' negative effect via a preliminary study, and then incorporate it into our designed reward function.

Figure 2(a) shows the overview of MCS. Next, we introduce the diversity measurement used in MCS in Section III-A. Then, we present our investigation of options' negative effect in Section III-B. Finally, we describe the whole process of memoized configuration search via multi-agent RL by utilizing the above two components in Section III-C.

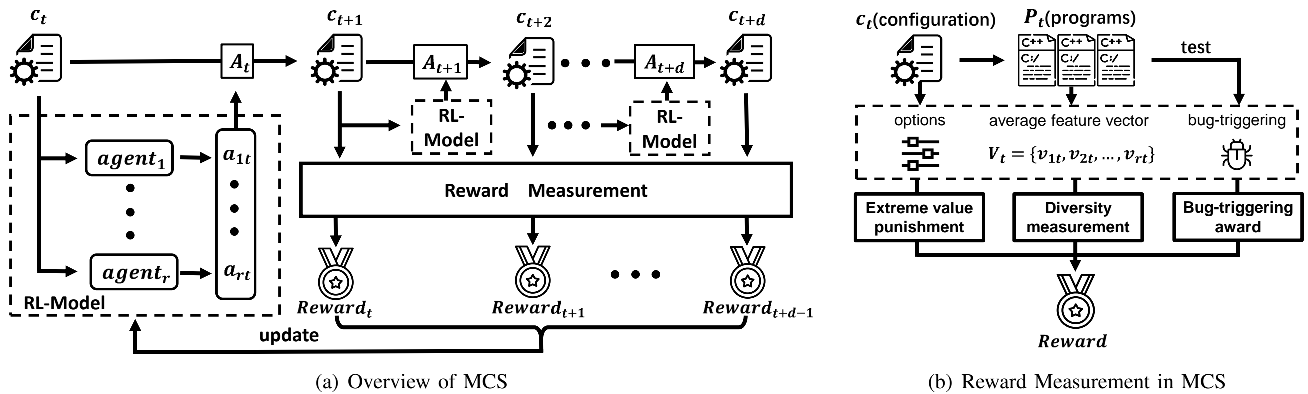
A. Diversity Measurement

Measuring diversity among test configurations aims to guide the search process for test configurations with different testing capabilities. The testing capability of a test configuration is actually embodied by the generated test programs under the configuration. Hence, following the existing work [13], MCS measures diversity among test configurations by measuring the diversity among *test programs* generated under different configurations. Specifically, MCS extracts the program features controlled by a test configuration from each test program, and represents them as a feature vector, each element in which is the number of occurrence times of the corresponding program feature in the program (e.g., the number of `goto` statements). There may be a portion of dead code in the program, but MCS does not distinguish them as dead code has been demonstrated to be also useful for compiler testing [21]. Then, MCS calculates diversity based on the feature vectors.

As a test configuration controls test-program generation in a probabilistic way, only one generated test program under the configuration cannot represent the testing capability of the configuration. Hence, by adopting the practice in the existing work [13], MCS uses a set of generated test programs under a configuration to statistically represent the testing capability of the configuration. As the generated test programs under a configuration tend to concentrate on an area of input space, MCS calculates the average feature vector of the set of test programs to represent the testing capability of the test configuration, and then measures the diversity based on those average feature vectors. In particular, to reduce the influence of different scales for different features, MCS processes the set of feature vectors for the generated test programs under a configuration via the widely-used standardization [38]. Then, MCS adopts cosine similarity to measure the distance between two test configurations (denoted as c_x and c_y), i.e., $\mathit{Dist}(c_x, c_y) = 1 - \mathit{cosine}(V_x, V_y)$ (V_x and V_y are corresponding average feature vectors for c_x and c_y). Please note that in the remaining of this paper, *the distance/diversity between test configurations refers to that between corresponding average feature vectors*.

B. Investigation on Options' Negative Effect

Regarding options' negative effect, we consider the generation of *the test programs running for an extremely long time*.



(a) Overview of MCS

(b) Reward Measurement in MCS

Fig. 2. An illustration of the framework and key component of MCS.

This is because as demonstrated in the existing study [22], such test programs have negative effect in the efficiency of compiler testing, which is the most important factor on the overall performance of compiler testing. Hence, effectively reducing the generation of such test programs is important. Here, we assume that the generation of such test programs is relevant to the extreme values of options. When setting an option to its extreme value, its controlled program feature has a quite large probability to be accepted at each decision point during the generation process. If there are a large number of options that are set to their extreme values, it is very likely to cause the combinatorial explosion of program features, leading to the lengthy generation time and the generation of very large-scale test programs that tend to run for an extremely long time. That is, the above negative effect could be aggravated as the number of options set to extreme values increases.

To investigate whether our assumption really holds, we conducted a preliminary study based on GCC-4.5.0 and Csmith. In the study, we set $w\%$ of options to the extreme values (referring to those larger than 95% and smaller than 5% of the upper bound in MCS), where we considered $w \in \{10, 20, \dots, 100\}$. For each w, we randomly selected $w\%$ of options, and for each of them we randomly selected a value from the extreme values as the setting of the option. Regarding the remaining options (except the selected $w\%$ of options), we randomly set their values (except the extreme values). After obtaining a test configuration setting, we generated and ran 100 test programs under the configuration, and recorded the time spent on each test program. Following the existing work [13], we treated the time of over 10 minutes as the long testing time, and *regarded those test programs with long testing time as negative-effect test programs*. For each w, we repeated the above process 100 times to obtain more significant results in statistics.

Figure 3 shows the result, in which the x-axis represents the value of w and the y-axis represents the average number of the generated test programs with long testing time across the 100 repeated experiments. We found that, as w's value increases, more test programs have long testing time in general, confirming our assumption to a large extent. The result indicates it is very necessary to consider how many options

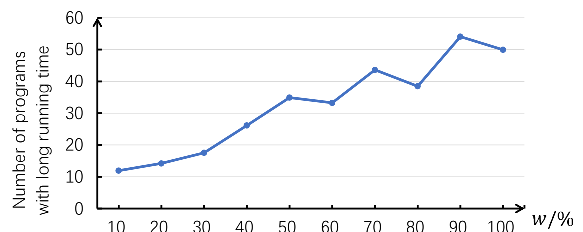


Fig. 3. Option's negative effect results.

are set to their extreme values during the search process of MCS for test configurations. Indeed, we consider this factor in our designed reward function (Section III-C). In current MCS, we incorporate options' negative effect from the view of the number of options set to their extreme values. There may be more advanced methods to consider the effect, and we discuss it in Section VII-B as our future work.

C. Multi-Agent RL based Test Configuration Construction

Based on the above components, MCS adopts multi-agent RL (i.e., multi-agent A2C) to guide the memoized-search process of test configurations during on-the-fly compiler testing. As it is important to learn the elaborate coordination among options and each option can be set individually, we assign each option an agent to separately and flexibly determine the action that it will take on each option at each time step during the learning process. Suppose a test configuration c is represented as $c = \{o_1, o_2, \dots, o_r\}$, where o_k $(1 \leq k \leq r)$ is the setting of the kth option in c and r is the number of options, and the agent for option o_k is denoted as $agent_k$, whose state is represented as s_{kt} and action that can be taken as $a_{kt} \in \mathcal{A}_k$ at a certain time step t. Then, the learning task is, for each $agent_k$, to predict a sequence of actions $\langle a_{k1}, a_{k2}, \dots, a_{kt}, \dots, a_{kT} \rangle$ on option o_k as correspondence, which can produce the largest cumulative reward regarding all the agents. As aforementioned that different options may have mutual effect, to capture such relations among them, we make the states of different agents shared. That is, when predicting the next action that an agent will take, it can observe the

states of all the agents (i.e., the settings of all the options), i.e, $\forall k \in [1, 2, \dots, r], s_{kt} = c_t = \{o_{1t}, o_{2t}, \dots, o_{rt}\}$, where s_{kt} denotes the state of $agent_k$ at time step t, while c_t denotes the whole test configuration at time step t. In this way, each agent can make the best choice of actions from a global view. In our scenario, different agents have the same set of actions $\mathcal{A}$, which refers to a set of operations on the current value of an option, including $+\delta$ (adding δ to the current value), $-\delta$ (subtracting δ from the current value), and keeping the current value unchanged. Please note that if the option's value exceeds its bounds, we set it to the corresponding boundary value.

Following the framework of A2C, MCS builds ANN and CNN for each $agent_k$ ($1 \leq k \leq r$). At a certain time step t, ANN predicts the probability distribution of actions based on learned knowledge from the memoization for explored configurations and then chooses an action a_{kt} for $agent_k$ to be performed, while CNN predicts the potential reward to be accumulated after applying a_{kt}. Finally, according to the predicted rewards and actual rewards obtained after applying predicted actions to $agent_k$ for all $k \in [1, 2, \dots, r]$, MCS updates ANN and CNN for each agent based on an advantage loss function, and then all the agents move to a new state, i.e., a new test configuration is constructed under the guidance of RL for the test-program generation. With the search process proceeding, more accurate ANN and CNN can be built based on the memoization for more explored configurations, and thus it can be more likely to generate an effective test configuration for improving compiler testing at each time step.

Next, we introduce the design of actual reward measurement and advantage loss function applied by MCS.

1) Actual Reward Measurement: A well-designed reward function is vital for RL, as it directly drives the learning process and guides the exploration of search space. Indeed, designing an effective reward function in RL to solve a specific task is challenging. In MCS, we consider three components in its reward function, including diversity component (discussed in Section III-A), extreme value punishment (discussed in Section III-B), and bug-triggering award. Figure 2(b) shows the overview of our actual reward measurement.

Diversity component. MCS considers the diversity among test configurations to more efficiently explore the input space, especially the space involving bug-triggering test programs. It is the key component in our designed reward function. The diversity of the current configuration c_t at time step t in MCS is measured by the average distance between c_t and the explored configurations. As presented in Section III-A, the distance/diversity between test configurations refers to that between *the corresponding average program-feature vectors*. As the explored configuration space could be larger and larger, there may be outliers that can dominate the average distance, and thus MCS calculates the average distance between c_t and a set of *closest* explored ones (rather than all the explored ones) as the approximation, which is defined as Formula 1.

$$div_t = \frac{1}{|C_h|} \sum_{c_i \in C_h} Dist(c_i, c_t) \tag{1}$$

where C_h is a set of explored test configurations that are closest to c_t, i.e., assuming C_t is the set of all the explored test configurations, $\forall c_i \in C_h, c_j \in C_t \setminus C_h$, $Dist(c_i, c_t) \leq Dist(c_j, c_t)$. $Dist(\cdot)$ is the distance between two test configurations (i.e., two corresponding average feature vectors for the sets of generated test programs) defined in Section III-A.

To avoid the exploration of test configurations falling into local optimum, we expect the configurations explored within a relatively short time interval are also diverse and the search process always proceeds towards a good direction. Hence, MCS further calculates the average improved diversity for c_t over m configurations with the closest time (i.e., learning steps) to c_t: $R_t^{div} = \frac{1}{m}\sum_{i=1}^{m}(div_t - div_{t-i})$. The average improved diversity is regarded as the diversity component in the reward function of MCS, which facilitates the efficient exploration of a wider range of test configurations.

Extreme value punishment. Although larger diversity helps explore diverse test configurations and leads to more bug-triggering test programs, it may increase the risk of producing unexpected ones, which could generate negative-effect test programs (i.e., those running for an extremely long time) for compiler testing. As explained in Section III-B, the number of options set to extreme values has correlations to the generation of negative-effect test programs. To avoid the generation of such test programs, we should give a punishment for the test configurations, in which many options are set to extreme values. Specifically, when there are more than $q\%$ options that are set to their extreme values in a test configuration, MCS gives a punishment λ to it, i.e., $R_t^{neg} = \lambda$, otherwise 0.

Bug-triggering award. Our goal is to generate bug-triggering test programs during the memoized search process for effective test configurations. The bug-triggering test programs also reveal the bug-triggering coordination among options, indicating that the corresponding portion of space is more likely to be bug-triggering and thus deserves more exploitation. Hence, if there are bug-triggering test programs generated under a configuration, we should give an award to it. We define the bug-triggering award in MCS as $R_t^{trg} = \omega * n_f$, where ω is a constant coefficient and n_f is the number of bug-triggering test programs generated under the configuration c_t.

With the above three components, we obtain the reward function in MCS: $Reward_t = R_t^{div} + R_t^{neg} + R_t^{trg}$.

2) Advantage Loss Function: At a certain time step t, the actual reward can be measured as above after applying the current test configuration c_t to generate a set of test programs. The corresponding estimated reward can be predicted by CNN. According to the actual and predicted rewards, we design our advantage loss functions for ANN and CNN, respectively. As ANN and CNN are independently maintained for each agent, to reduce computational overhead and boost the learning procedure, we adopt a *d-step* updating strategy, which has been demonstrated to be efficient and effective by the existing study [19]. That is, ANN and CNN will be updated each time after *d* consecutive actions are predicted for all agents. Formulae 3 and 4 define the advantage loss functions for ANN

and CNN after each round of action prediction for $agent_k$ respectively, where $1 \leq k \leq r$ and r is the number of agents.

$$R_k(t) = Reward_t + \gamma R_k(t+1) \quad (2)$$

$$\mathcal{L}_k^{ANN}(t) = logP_{\theta_k}(s_{k(t+1)}|s_{kt}, a_{kt})(R_k(t) - V_{\varphi_k}(s_{kt})) \quad (3)$$

$$\mathcal{L}_k^{CNN}(t) = (R_k(t) - V_{\varphi_k}(s_{kt}))^2 \quad (4)$$

In these formulae, $P_{\theta_k}(s_{k(t+1)}|s_{kt}, a_{kt})$ denotes the *transition probability* of choosing action a_{kt} under state s_{kt}, which is predicted by ANN under parameter θ_k, while $V_{\varphi_k}(s_t)$ denotes the predicted reward by CNN under parameter φ_k. γ is a global discount factor for incorporating current and future rewards. Hence, at each time step t_0 per *d-step* (i.e., $t_0 = d * n, n \in \mathbb{N}$), the network parameters of all agents will be updated by the accumulated gradients accordingly based on Formulae 5 and 6, where η is the learning rate. Here, $R_k(t_0+d) = V_{\varphi_k}(s_{t_0+d})$ for each t_0 for the computation of Formula 2.

$$\theta_k = \theta_k + \eta \sum_{t=t_0}^{t_0+d} \frac{\partial \mathcal{L}_k^{ANN}(t)}{\partial \theta_k} \quad (5)$$

$$\varphi_k = \varphi_k + \eta \sum_{t=t_0}^{t_0+d} \frac{\partial \mathcal{L}_k^{CNN}(t)}{\partial \varphi_k} \quad (6)$$

IV. Experimental Study Design

In the study, we address the following research questions:

- **RQ1**: How does MCS perform in detecting compiler bugs compared with state-of-the-art approaches?
- **RQ2**: How does MCS perform under different configurations?

A. Subjects

In our study, we used two popular C compilers as subjects, i.e., GCC and LLVM, following the existing studies [11], [17], [21], [22], [39]. Specifically, we adopted the widely-studied versions in existing compiler-testing studies, including three versions of GCC (i.e., GCC-4.4.0, GCC-4.5.0, and GCC-4.6.0) and two versions of LLVM (i.e., LLVM-2.8 and LLVM-4.0) for the x86 64-Linux platform. Here, we used these historical versions as they usually contain more bugs and can provide more significant results in statistics. To investigate whether MCS still works on newer versions, we also applied it and compared approaches to test the latest trunk revisions of both GCC and LLVM[1] following the existing studies [21], [40].

B. Tools and Configurations

In our study, we used the most widely-used test-program generator, i.e., Csmith [11], as the studied one. Its test configuration contains 71 options. It takes a configuration file that provides the value of each option as input and produces a C program according to the configuration as output. Csmith has some heuristics and safety checks to avoid undefined behaviors. Its generated test program does not require external inputs and its output is a checksum of the non-pointer global variables at the end of program execution.

We implemented MCS atop PyTorch [41]. We set the default settings of MCS's parameters based on a small experiment: the sizes of test configurations considered for diversity measurement are $|C_h|$=10 and m=10; the step for model updating is *d-step*=10; the threshold and punishment of extreme options are *q*%=30% and λ=-2; the coefficient in bug-triggering award is ω=4; and the action of δ is 5, the discount factor γ is 0.9, the learning rate is 0.01. Also, Csmith generates 100 test programs under each configuration for testing. We investigated the influence of main parameters on the performance of MCS in Section V-B.

To determine whether a test program triggers a compiler bug, we adopted differential testing [24] as the test oracle. If a test program produces different outputs after execution under different optimizations of a compiler (e.g., `-O0`, `-O1`, `-O2`, `-O3`, and `-Os` in GCC), it means it triggers a bug in the compiler. Here, we left out non-terminating test programs by setting a timeout (i.e., 60 seconds).

All the experiments were conducted on a workstation with a 72-core CPU, 126G memory, and CentOS Linux release 7.8 operating system. In particular, *it took us over eight months to run our experiments.*

C. Compared Approaches

MCS is a test-program generation approach via memoized configuration search, and thus we compared it with the existing approaches that also explore test configurations for better test-program generation. As presented in Section II-A, there are two categories of approaches, and thus we chose the state-of-the-art approach in each category for comparison, i.e., HiCOND and swarm testing (abbreviated as Swarm in this paper). **HiCOND** is an offline approach, which infers a set of test configurations by mining historical bugs [13]. As the implementation of HiCOND is open-source, we directly adopted their implementation. **Swarm** is an online approach, which randomly generates a test configuration before the generation of each test program [14]. Specifically, it randomly assigns a valid value to each option in a test configuration.

In RQ2, we first investigated whether extreme value punishment and bug-triggering award in our reward function are helpful for compiler testing. To answer it, we constructed two variants for comparison, i.e., **MCS**$^-$ removing the item of R_t^{neg} from the reward function in MCS and **MCS**$^+$ removing the item of R_t^{trg}. We did not construct the variant removing the diversity component as it is the key component in MCS. Then, we investigated the influence of two main parameters in MCS, i.e., $|C_h|$ and m for diversity measurement, by studying $|C_h| \in \{5, 10, 15, 20\}$ and $m \in \{5, 10, 15, 20\}$, respectively. In this RQ, we keep the default settings for other parameters.

D. Measurements

Following the existing work [13], [17], we used the number of detected bugs within the same testing period to measure the performance of test-program generation approaches. Same

[1]The IDs of trunk revisions under test are from 1afa4fa to bb6194e for GCC and from a048ce1 to ae0e037 for LLVM.

TABLE I
NUMBER OF DETECTED BUGS

Approach	GCC 4.4.0	GCC 4.5.0	GCC 4.6.0	LLVM 2.8	LLVM 4.0	*Total*
MCS	**38**	**17**	**4**	**13**	**2**	**74**
HiCOND	16	8	3	8	1	36
Swarm	13	0	0	5	0	18

TABLE II
DETAILS OF DETECTED NEW BUGS

Subject	ID	Symptom	Component	Status
GCC	97980	wrong code	tree-optimization	Confirmed
GCC	99296	crash	tree-optimization	Fixed
GCC	101080	wrong code	tree-optimization	Fixed
GCC	101062	wrong code	middle-end	Fixed
GCC	101009	wrong code	tree-optimization	Fixed
GCC	101249	crash	tree-optimization	Fixed
GCC	101594	crash	rtl-optimization	Confirmed
GCC	102565	crash	tree-optimization	Fixed
GCC	102627	wrong code	rtl-optimization	Fixed
LLVM	48778	wrong code	—	Fixed
LLVM	48812	crash	loop-optimization	Fixed
LLVM	48871	crash	—	Confirmed
LLVM	49105	wrong code	—	Fixed
LLVM	51903	crash	back-end	Fixed
LLVM	51907	wrong code	—	Fixed
LLVM	52023	crash	—	Fixed

as the existing study [13], we tested each subject using each approach for 10 days, and adopted Correcting Commit [22] for de-duplication to identify the number of detected bugs from a set of bug-triggering test programs. Specifically, for each bug-triggering test program, it searches for the first commit making the program pass. If the same correcting commit is found for two bug-triggering test programs, we regard that the two programs trigger the same bug. This method may be a threat to our study. However, it is the only automatic method to measure the number of detected bugs and its accuracy has been demonstrated by the existing study [22], and thus this threat may be not serious.

In addition, we applied these approaches to test the latest trunk revisions of both GCC and LLVM. All the bugs detected in this experiment are new bugs, and thus Correcting Commit is not applicable. For these bugs, we submitted bug reports to developers, and then determine the number of detected new bugs according to the developers' feedback.

E. Process

To answer RQ1, we applied MCS, HiCOND, Swarm to test each subject for 10 days, respectively. During the testing process, we recorded the testing result for each test program.

To answer RQ2, we applied MCS$^-$ and MCS$^+$, as well as MCS with different configurations of $|C_h|$ and m to test each subject, respectively. As the testing process lasts 10 days each time, leading to huge time cost on running them for all the subjects, in this RQ we used GCC-4.5.0 and LLVM-2.8 as the representative. Similarly, we recorded the testing result for each test program.

V. RESULTS AND ANALYSIS

A. RQ1: Performance of MCS

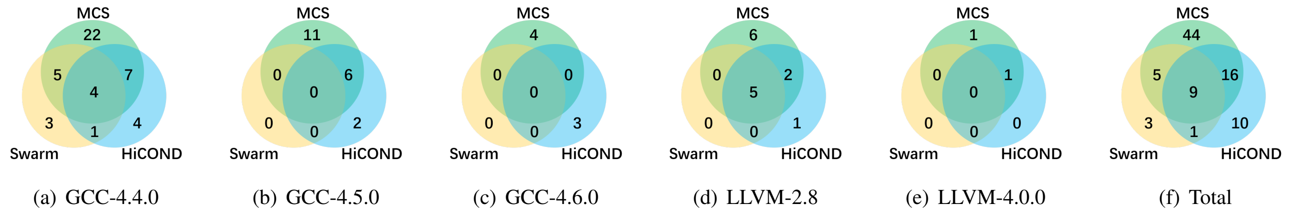
1) Number of Detected Bugs: Table I shows the number of detected bugs by each studied test-program generation approach during the same testing period (i.e., 10 days). We found that MCS detects the largest number of bugs among the three approaches on each subject. In total, MCS detects 74 bugs while HiCOND and Swarm detect 36 and 18 bugs respectively. The improvements of the former over the latter two are 105.56% and 311.11% respectively, demonstrating the superiority of MCS in detecting compiler bugs.

We further analyzed various relations among the bugs detected by the three approaches. We used Venn diagrams as shown in Figure 4 to illustrate the relations. For example, in Figure 4(a) the number of unique bugs (which refer to the bugs that can be detected by only one approach) detected by MCS, HiCOND, and Swarm on GCC-4.4.0 is 22, 4 and 3, respectively. In particular, Figure 4(f) shows the overall results for all the subjects. We found that MCS detects the largest number of unique bugs, i.e., 44, while HiCOND and Swarm detect 10 and 3 unique bugs, respectively. In particular, among the bugs detected by MCS, 59.46% (44 out of 74) of them are unique. The results demonstrate the significant unique value of MCS in detecting compiler bugs.

HiCOND and Swarm can detect some unique bugs, indicating these approaches can complement each other to some degree. The reason why HiCOND detects unique bugs may be that it uses a fixed set of configurations inferred offline for compiler testing, which makes it focus on a certain portion of input space. During the given testing period, it can exploit the portion of space more sufficiently, while MCS explores larger input space in an online way and thus may not sufficiently exploit the space focused by HiCOND, resulting in the missing of some bugs. Regarding the unique bugs detected by Swarm, the reason mainly lies in it can explore a certain portion of input space missed by MCS due to no guidance during the search process of Swarm. As only 4 bugs detected by Swarm are missing to be detected by MCS, it indicates our guided search process is relatively sufficient for compiler testing. Overall, MCS reaches a good balance between exploration and exploitation of the input space during a given testing period.

Besides, we found that all the approaches have the same phenomenon (i.e., effectiveness drops on GCC-4.6.0), and the reason may be this version has been immune to the underlying Csmith to some degree. The similar phenomenon was also observed in the existing work [40]. Nevertheless, MCS still outperforms the compared approaches and detects many new bugs in the latest revisions of GCC and LLVM (presented in Section V-A2), demonstrating that MCS could further activate the bug-triggering capability of the underlying tool.

2) Number of New Bugs Detected by MCS: Following the existing studies [21], [40], we applied MCS to test the



(a) GCC-4.4.0 (b) GCC-4.5.0 (c) GCC-4.6.0 (d) LLVM-2.8 (e) LLVM-4.0.0 (f) Total

Fig. 4. Number of unique bugs.

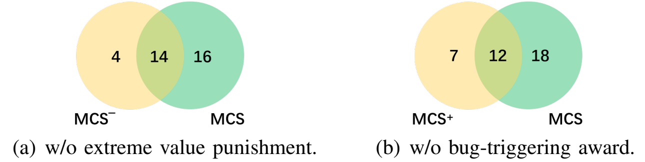
```
1 struct {
2  signed a : 1;
3 } b, c;
4 void d() {
5  b.a|=c.a|=0!=2; }
6 int main() {}
```

(a) GCC bug (ID: 99296)

```
1 static int a = -1L;
2 int b;
3 int main() {
4  b=a>0&&(-214748364
       -1)/a?0:a;
5 }
```

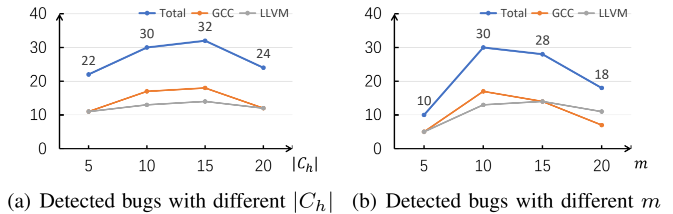
(b) LLVM bug (ID: 48778)

Fig. 5. Example test programs that trigger new bugs.



(a) w/o extreme value punishment. (b) w/o bug-triggering award.

Fig. 6. Comparison between MCS and its variants.



(a) Detected bugs with different $|C_h|$ (b) Detected bugs with different m

Fig. 7. Results of MCS under different configurations.

latest trunk revisions of GCC and LLVM for three months, to investigate whether MCS works on the latest compiler versions. Table II shows the details of detected new bugs by MCS, including bug ID, bug symptom, bug-occurring compiler component labeled by developers, and bug status. In total, MCS successfully detected 9 new bugs from GCC and 7 new bugs from LLVM, all of which have been confirmed or fixed by developers. These bugs involve both crash and wrong code (producing unexpected outputs without crash) symptoms in several different compiler components, confirming that MCS can detect diverse bugs. Although most existing studies do not run compared approaches on the latest compiler revisions for comparison due to the very long running time [21], [40], we still applied HiCOND and Swarm to the same revisions for sufficient comparison. As this experiment is too costly, we ran each compared approach for one month, and we found both Swarm and HiCOND cannot detect any bugs on the same revisions, but MCS detects 5 bugs (all of them have been fixed) during *one-month* testing. The possible reason for HiCOND is that its inferred configurations based on historical bugs are limited on the latest revisions, while that for Swarm lies in its low efficiency for exploring the huge space.

To better understand the detected bugs, we present two example bugs detected by MCS in Figure 5 (we simplified the bug-triggering test programs and conditions to facilitate developers' understanding). In the first case (Figure 5(a)), the latest GCC-11.0.1 compiler failed to compile it with the options of "`-fno-tree-bit-ccp -Os`" due to signed 1-bit precision by the code of "`a:1`". This bug is critical with the severity of "P1" (the highest severity in GCC), and is regarded as "*nightmare to deal with*" by the developer. Similarly, the second test program (Figure 5(b)) crashed the compiling process with LLVM-12.0.0, which is caused by an incorrect `if` condition for checking if a denominator is "-1". The results further demonstrate the performance of MCS.

B. RQ2: Performance of Different MCS's Configurations

We first investigated the contribution of extreme value punishment in MCS by comparing it with MCS$^-$. Figure 6(a) shows the results. From this figure, 16 bugs detected by MCS cannot be found by MCS$^-$, indicating the necessity of considering extreme value punishment. Then, we explored the contribution of the bug-triggering award by comparing MCS with MCS$^+$. The result is shown in Figure 6(b). From Figure 6(b), the bug-triggering award contributes to 18 more unique bugs detected by MCS, demonstrating its significant contribution. However, MCS$^-$ and MCS$^+$ detected additional 4 and 7 bugs that MCS fails to detect, respectively. The results reflect the effectiveness of the diversity measurement in MCS, which can still guide the search process and lead to a suboptimal result even without incorporating extreme value punishment and bug-triggering award. In summary, both components largely contribute to the performance of MCS.

Finally, we explored the influence of $|C_h|$ and m, which affect the diversity measurement, one key component in MCS. We conducted a series of controlled experiments for each setting of $|C_h| \in \{5, 10, 15, 20\}$ and $m \in \{5, 10, 15, 20\}$. Figure 7 shows the total number of bugs detected on the studied compilers (i.e., GCC-4.5.0 and LLVM-2.8). From the figure, although the results vary under different configurations, MCS almost always outperforms the compared approaches as shown in Table I (HiCOND detected 8 and Swarm detected 5 bugs on GCC-4.5.0 and LLVM-2.8), except for the case where $m = 5$. The results further demonstrate MCS is more effective

for exploring diverse test configurations that lead to bug-triggering test programs than existing approaches. Regarding the performance drop when $m = 5$, the major reason is that too small m can make MCS fall into local optimal as only limited number of local configurations considered can be insufficient. In summary, MCS stably outperforms existing approaches under different configurations, and our default settings for the parameters are indeed proper, making MCS promising for practical use with little effort on determining its parameters.

VI. Industry Application

MCS has been successfully deployed by a global IT company (i.e., *Huawei*) to test their in-house compiler, which is a high-performance and easy-to-expand C compiler for general-purpose processor architectures. Due to the company policy, we hide the compiler name. For ease of presentation, we call the compiler $\mathcal{C}$. $\mathcal{C}$ has been developed for over two years and six versions have been released in *Huawei*. The scale of $\mathcal{C}$ is more than 6 million SLOC (source lines of code). Many of their products are built on top of this compiler, and thus guaranteeing its quality is an urgent demand for them.

We reported the results of the industry evaluation by deploying MCS to test the latest version of $\mathcal{C}$ for ten days. In total, MCS detects 10 bugs and all of them have been confirmed by developers. These bugs include 1 *crash* bug that makes the compiler crash when compiling the test program, 9 *wrong code* bugs that produce inconsistent outputs when compiling and executing the test programs under different optimizations. We also applied the compared approaches (i.e., HiCOND and Swarm) to test $\mathcal{C}$ for ten days. HiCOND detects 5 bugs and Swarm detects 4 bugs, and all of these bugs are detected by MCS. However, 5 bugs detected by MCS cannot be detected by both HiCOND and Swarm during the same testing time. The results confirm the practical value of MCS.

VII. Discussion

A. RL vs Conventional Search Algorithms

MCS adopts the state-of-the-art multi-agent RL to capture implicit and complex coordination among options for guiding the effective generation of test programs. Actually, there are some conventional search algorithms (e.g., Genetic Algorithm [42] and PSO [43]) that can be also adopted to fit our task. Hence, we investigated whether using such simple algorithms is enough or adopting advanced RL is necessary.

To answer it, we implemented a variant of MCS (called MCS_{pso}), which uses PSO to guide the online search process instead of RL by taking our reward function as its fitness function. The reason why we chose PSO is that it is effective to search in a continuous space [43] and has been used in some testing tasks [13], [44], [45]. Following the framework of PSO, MCS_{pso} treats a test configuration as a particle, and guides a set of particles to fly towards the direction of larger fitness values. For fair comparison, MCS_{pso} keeps the same settings as MCS except the search algorithm. Regarding the parameters in PSO, it uses the settings provided by the existing work [13], [44]. Then, we conducted an experiment to compare MCS with MCS_{pso} by taking GCC-4.5.0 and LLVM-2.8 as the representative (same as Section V-B). During 10-day testing, MCS detects 17 and 13 bugs on GCC-4.5.0 and LLVM-2.8.0 respectively, while MCS_{pso} detects 4 and 8 bugs, demonstrating incorporating state-of-the-art RL makes significant contributions to the performance of MCS.

Besides, we compared RL in MCS with the conventional machine learning algorithm by adapting the XGBoost algorithm [46] to an online learning style. The results also confirm the significant superiority of RL. Due to the space limit, we put the comparison details at our project homepage [47].

B. Extension of MCS

First, we investigated the options' negative effect via a preliminary study and considered it in our reward function according to the finding. Also, multi-agent RL can implicitly learn mutual effects among options via interactions among agents. In the future, we can adopt machine learning to model mutual effects *more systematically* (e.g., building a model to predict whether a configuration can mostly generate negative-effect test programs), which may avoid low-quality configurations more sufficiently and thus further improve MCS.

Second, we will extend MCS to other software with complex test inputs in the future. Although MCS is proposed and evaluated based on C compilers, *there is no characteristic in MCS specific to C compilers*. That is, the idea of MCS is general. Applying it to other software has two conditions: 1) there are test input generators that can be controlled by a test configuration; 2) there are features that can be extracted from test inputs relevant to the options in the configuration. Hence, it is easy to extend MCS to the software with complex test inputs (e.g., the compilers for other programming languages and browsers), as it is very likely for their test inputs to contain various features and there are many such fuzzing tools for them, e.g., CLsmith [12] for OpenCL compilers and jsfunfuzz [48] for JavaScript engines.

Besides, the current implementation of MCS is based on Csmith and it may inherit the limitations of it. For example, it cannot generate invalid test programs and thus can miss to detect some bugs as demonstrated by the existing work [49]. In the future, we can extend MCS on the test generator that can generate invalid test programs, to further improve its effectiveness.

C. Threats to Validity

The threats to *internal* validity mainly lie in the implementations of MCS and experimental scripts. To reduce this kind of threat, two authors carefully checked all our code via code review and designing test cases. In our study, we found there are some differences on the performance of HiCOND and Swarm between our obtained results and the results reported by their original papers. The main reasons lie in 1) different running environments, and 2) more strictly filtering out undefined behaviors as suggested by compiler developers.

The threat to *external* validity mainly lies in the used test-program generator. We used Csmith following the existing

work [13], [14]. Although it may not represent other tools, this kind of threat may be not serious as 1) Csmith is the most widely-used one in C compiler testing; 2) It is the only test-program generator used in the existing studies [13]–[15]; 3) Many recent tools are adapted from it, such as CLsmith [12].

The threats to *construct* validity mainly lie in randomness and parameters' settings. There exists randomness caused by the test-program generator and the inherent characteristics of these approaches. Following the existing studies [13], [22], we used a long testing period to reduce this threat instead of repeating our experiments several times. During the long testing period, MCS generates 78,637 to 151,922 test programs for these subjects. To reduce the threat from the parameters' settings in MCS, we presented the specific settings in Section IV-B and investigated their influence in Section V-B. Based on our results, the current settings make MCS achieve stable effectiveness across different subjects, and thus we released them as the default settings in MCS.

VIII. Related Work

Compiler Testing. Over the years, many compiler testing research projects have been conducted [3], [12], [15], [50]–[53]. The most related to our work is test-program generation, especially those based on test configurations. Besides our studied Csmith [11] and our compared approaches, i.e., HiCOND [13] and swarm testing [14], Rabin and Alipour [16] extracted insights from historical bug reports about error-prone language features to produce test configurations, in order to guide test-program generation. Also, Alipour et al. [15] proposed directed swarm testing that uses statistics and a variation of random testing to produce test programs focusing on a given compiler code element. We did not compare with it as its purpose of producing test programs is to increase the covering frequency for a given small part of compiler code, which is not aligned with ours, i.e., sufficiently exploring the whole input space.

Besides, Le et al. [21] introduced *equivalent modulo inputs* (EMI) for testing compilers, which produces equivalent test programs under certain test inputs. MCS is orthogonal to EMI-based approaches, as MCS can provide original test programs and then EMI-based approaches can produce equivalent programs based on them for compiler testing. MCS is also orthogonal to grammar-based test-program generators (if they rely on test configurations to generate test programs like Csmith), as it can help them construct better configurations.

RL in Software Testing. RL has gained great success in many scenarios [54]–[62], and researchers also applied them in software testing tasks. For example, Kim et al. [63] presented QTIP that uses Q-learning to generate test inputs for small C programs. Recently, RL was used for testing some domain-specific applications [64]–[70]. For example, Pan et al. [71] proposed a curiosity-driven exploration strategy based on RL to guide automated Android testing. Pan et al. [62] proposed a delay-based hardware Trojan detection method, which uses RL to generate tests for triggering rare switches. Different from them, we are the first to employ RL for *compiler test-program generation*. In particular, we incorporated the multi-agent framework of A2C in MCS, different from their used RL algorithms.

IX. Conclusion

We proposed a novel compiler test-program generation approach, called MCS, which conducts memoized search via multi-agent RL for guiding the construction of effective test configurations by incorporating three kinds of configuration characteristics, i.e., diversity, options' negative effects, and the ability to produce bug-triggering programs, for reward measurement. Our results demonstrated MCS largely outperforms the state-of-the-art compared approaches. In particular, MCS detected 16 new GCC and LLVM bugs, all of which have been confirmed/fixed by developers. MCS has been deployed by *Huawei* for testing their in-house compiler.

Acknowledgment

We thank the anonymous reviewers for their constructive suggestions to help improve the quality of this paper. This work was supported by the National Natural Science Foundation of China under Grant Nos. 62002256, 62232001, 62202324, and Huawei Project Funding.

References

[1] X. Li, J. Jiang, S. Benton, Y. Xiong, and L. Zhang, "A large-scale study on api misuses in the wild," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2021, pp. 241–252.

[2] J. Jiang, Y. Xiong, and X. Xia, "A manual inspection of defects4j bugs and its implications for automatic program repair," *Science China Information Sciences*, vol. 62, p. 200102, Sep 2019.

[3] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," *ACM Computing Surveys*, vol. 53, no. 1, pp. 1–36, 2020.

[4] V. Livinskii, D. Babokin, and J. Regehr, "Random testing for c and c++ compilers with yarpgen," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–25, 2020.

[5] A. F. Donaldson, H. Evrard, and P. Thomson, "Putting randomized compiler testing into production (experience report)," in *34th European Conference on Object-Oriented Programming*, 2020.

[6] M. Marcozzi, Q. Tang, A. F. Donaldson, and C. Cadar, "Compiler fuzzing: How much does it matter?" *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[7] I. Hussain, C. Csallner, M. Grechanik, Q. Xie, S. Park, K. Taneja, and B. Mainul Hossain, "Rugrat: Evaluating program analysis and testing tools and compilers with large generated random benchmark applications," *Software: Practice and Experience*, vol. 46, no. 3, pp. 405–431, 2016.

[8] S. A. Chowdhury, S. L. Shrestha, T. T. Johnson, and C. Csallner, "Slemi: Equivalence modulo input (emi) based mutation of cps models for finding compiler bugs in simulink," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 335–346.

[9] K. Even-Mendoza, C. Cadar, and A. F. Donaldson, "Closer to the edge: Testing compilers more thoroughly by being less conservative about undefined behaviour," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1219–1223.

[10] Y. Zhao, Z. Wang, J. Chen, M. Liu, M. Wu, Y. Zhang, and L. Zhang, "History-driven test program synthesis for jvm testing," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1133–1144.

[11] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294.

[12] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015, pp. 65–76.

[13] J. Chen, G. Wang, D. Hao, Y. Xiong, H. Zhang, and L. Zhang, "History-guided configuration diversification for compiler test-program generation," in *34th IEEE/ACM International Conference on Automated Software Engineering*, 2019, pp. 305–316.

[14] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr, "Swarm testing," in *International Symposium on Software Testing and Analysis*, 2012, pp. 78–88.

[15] M. A. Alipour, A. Groce, R. Gopinath, and A. Christi, "Generating focused random tests using directed swarm testing," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 70–81.

[16] M. R. I. Rabin and M. A. Alipour, "Configuring test generators using bug reports: a case study of GCC compiler and csmith," in *SAC '21: The 36th ACM/SIGAPP Symposium on Applied Computing*, 2021, pp. 1750–1758.

[17] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie, "Learning to prioritize test programs for compiler testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering*, 2017, pp. 700–711.

[18] C. Sun, V. Le, Q. Zhang, and Z. Su, "Toward understanding compiler bugs in gcc and llvm," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 294–305.

[19] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*. PMLR, 2016, pp. 1928–1937.

[20] J. Chen, H. Ma, and L. Zhang, "Enhanced compiler bug isolation via memoized search," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, p. 78–89.

[21] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proceedings of the 35th ACM SIGPLAN conference on Programming language design and implementation*, 2014, pp. 216–226.

[22] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "An empirical comparison of compiler testing techniques," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 180–190.

[23] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 197–208.

[24] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[25] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, "A survey on metamorphic testing," *IEEE Transactions on software engineering*, vol. 42, no. 9, pp. 805–824, 2016.

[26] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, "Compiler fuzzing through deep learning," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 95–105.

[27] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[28] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[29] C. Zheng, S. Yang, J. M. Parra-Ullauri, A. Garcia-Dominguez, and N. Bencomo, "Reward-reinforced generative adversarial networks for multi-agent systems," *IEEE Transactions on Emerging Topics in Computational Intelligence*, 2021.

[30] J. Chen, N. Xu, P. Chen, and H. Zhang, "Efficient compiler autotuning via bayesian optimization," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1198–1209.

[31] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois *et al.*, "Milepost gcc: Machine learning enabled self-tuning compiler," *International journal of parallel programming*, vol. 39, no. 3, pp. 296–327, 2011.

[32] S. Fang, W. Xu, Y. Chen, L. Eeckhout, O. Temam, Y. Chen, C. Wu, and X. Feng, "Practical iterative optimization for the data center," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 2, pp. 1–26, 2015.

[33] A. H. Ashouri, G. Mariani, G. Palermo, E. Park, J. Cavazos, and C. Silvano, "Cobayn: Compiler autotuning framework using bayesian networks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 2, pp. 1–25, 2016.

[34] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "{TVM}: An automated {End-to-End} optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.

[35] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[36] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in neural information processing systems*, 2000, pp. 1057–1063.

[37] L. Panait and S. Luke, "Cooperative multi-agent learning: The state of the art," *Autonomous agents and multi-agent systems*, vol. 11, no. 3, pp. 387–434, 2005.

[38] G. W. Milligan and M. C. Cooper, "A study of standardization of variables in cluster analysis," *Journal of classification*, vol. 5, no. 2, pp. 181–204, 1988.

[39] J. Chen, J. Han, P. Sun, L. Zhang, D. Hao, and L. Zhang, "Compiler bug isolation via effective witness test program generation," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 223–234.

[40] J. Chen and C. Suo, "Boosting compiler testing via compiler optimization exploration," *ACM Transactions on Software Engineering and Methodology*, 2022.

[41] "Pytorch," Accessed: 2021, https://pytorch.org.

[42] D. Whitley, "A genetic algorithm tutorial," *Statistics and computing*, vol. 4, no. 2, pp. 65–85, 1994.

[43] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95-international conference on neural networks*, vol. 4, 1995, pp. 1942–1948.

[44] J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, and H. Mei, "Search-based inference of polynomial metamorphic relations," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 701–712.

[45] B. Zhang, H. Zhang, J. Chen, D. Hao, and P. Moscato, "Automatic discovery and cleansing of numerical metamorphic relations," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 235–245.

[46] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[47] "MCS," 2022, https://github.com/tju-chenyaosuo/MCS.

[48] "jsfunfuzz," Accessed: 2021, https://github.com/MozillaSecurity/funfuzz.

[49] H. Zhong, "Enriching compiler testing with real program from bug report," in *2022 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[50] A. S. Boujarwah and K. Saleh, "Compiler test case generation methods: a survey and assessment," *Information and software technology*, vol. 39, no. 9, pp. 617–625, 1997.

[51] Q. Shen, H. Ma, J. Chen, Y. Tian, S.-C. Cheung, and X. Chen, "A comprehensive study of deep learning compiler bugs," in *Proceedings of the 29th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2021, pp. 968–980.

[52] J. Chen, G. Wang, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "Coverage prediction for accelerating compiler testing," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 261–278, 2018.

[53] Q. Zhang, C. Sun, and Z. Su, "Skeletal program enumeration for rigorous compiler testing," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, p. 347–361.

[54] B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. Al Sallab, S. Yogamani, and P. Pérez, "Deep reinforcement learning for autonomous driving: A survey," *IEEE Transactions on Intelligent Transportation Systems*, 2021.

[55] X. Huang, T. Yuan, G. Qiao, and Y. Ren, "Deep reinforcement learning for multimedia traffic control in software defined networking," *IEEE Network*, vol. 32, no. 6, pp. 35–41, 2018.

[56] J. Koo, C. Saumya, M. Kulkarni, and S. Bagchi, “Pyse: Automatic worst-case test generation by reinforcement learning,” in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 136–147.
[57] R. Gupta, A. Kanade, and S. Shevade, “Deep reinforcement learning for syntactic error repair in student programs,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019, pp. 930–937.
[58] Y. Chen, C. Wang, O. Bastani, I. Dillig, and Y. Feng, “Program synthesis using deduction-guided reinforcement learning,” in *International Conference on Computer Aided Verification*, 2020, pp. 587–610.
[59] M. Sridharan and G. Tesauro, “Multi-agent q-learning and regression trees for automated pricing decisions,” in *Game theory and decision theory in agent-based systems*, 2002, pp. 217–234.
[60] M. Bagherzadeh, N. Kahani, and L. Briand, “Reinforcement learning for test case prioritization,” *IEEE Transactions on Software Engineering*, 2021.
[61] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, “Reinforcement learning for automatic test case prioritization and selection in continuous integration,” in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017, New York, NY, USA, 2017, p. 12–22. [Online]. Available: https://doi.org/10.1145/3092703.3092709
[62] Z. Pan, J. Sheldon, and P. Mishra, “Test generation using reinforcement learning for delay-based side-channel analysis,” in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2020, pp. 1–7.
[63] J. Kim, M. Kwon, and S. Yoo, “Generating test input with deep reinforcement learning,” in *2018 IEEE/ACM 11th International Workshop on Search-Based Software Testing (SBST)*, 2018, pp. 51–58.
[64] J. Bergdahl, C. Gordillo, K. Tollmar, and L. Gisslén, “Augmenting automated game testing with deep reinforcement learning,” *2020 IEEE Conference on Games (CoG)*, pp. 600–603, 2020.
[65] J. Eskonen, J. Kahles, and J. Reijonen, “Automating gui testing with image-based deep reinforcement learning,” in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, 2020, pp. 160–167.
[66] Y. Koroglu, A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi, and Y. Donmez, “Qbe: Qlearning-based exploration of android applications,” in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 2018, pp. 105–115.
[67] D. Adamo, M. K. Khan, S. Koppula, and R. Bryce, “Reinforcement learning for android gui testing,” in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2018, p. 2–8.
[68] T. A. T. Vuong and S. Takada, “A reinforcement learning based approach to automated testing of android applications,” in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2018, p. 31–37.
[69] A. Romdhana, A. Merlo, M. Ceccato, and P. Tonella, “Deep reinforcement learning for black-box testing of android apps,” *arXiv preprint arXiv:2101.02636*, 2021.
[70] Y. Zheng, X. Xie, T. Su, L. Ma, J. Hao, Z. Meng, Y. Liu, R. Shen, Y. Chen, and C. Fan, “Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 772–784.
[71] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, “Reinforcement learning based curiosity-driven testing of android applications,” in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020, New York, NY, USA, 2020, p. 153–164. [Online]. Available: https://doi.org/10.1145/3395363.3397354