Compiler Test-Program Generation via Memoized Configuration Search

Junjie Chan, Chenyao Suo, Jiajun Jiang, Peiqi Chen, Xingjian Li

2024.04.25.

Compiler and Compiler Test

- Compilers are one of the most fundamental software

Compiler and Compiler Test

- Compilers are one of the most fundamental software

Program (C)

```
int add(int a, int b) {
    return a + b;
}
```

Compiler and Compiler Test

- Compilers are one of the most fundamental software

Program (C)

```c
int add(int a, int b) {
    return a + b;
}
```

Program (Asm)

```
add %rdx, %rcx
mov %rdx, %rax
ret
```

Compiler and Compiler Test

- Compilers are one of the most fundamental software

Program (C)

```c
int add(int a, int b) {
    return a + b;
}
```

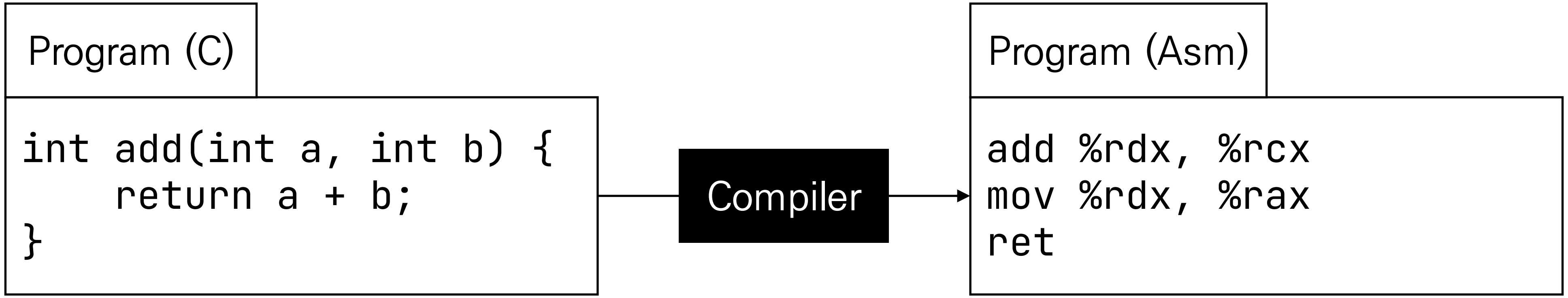
Compiler

Program (Asm)

```
add %rdx, %rcx
mov %rdx, %rax
ret
```

Compiler and Compiler Test

- Compilers are one of the most fundamental software



Bugs in Compiler

- What if the compiler has bugs?

Program (C)

```c
int add(int a, int b) {
    return a + b;
}
```

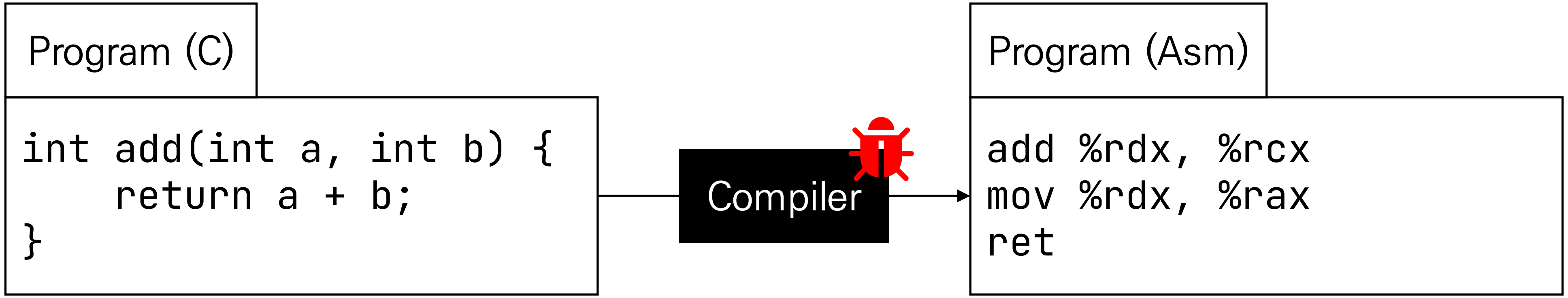
Compiler

Program (Asm)

```
add %rdx, %rcx
mov %rdx, %rax
ret
```

Problem: Bugs in Compiler

- What if the compiler has bugs?



Problem: Bugs in Compiler

- What if the compiler has bugs?

Program (C)

```c
int add(int a, int b) {
    return a + b;
}
```

Compiler

Program (Asm)

```asm
add %rcx, %rdx
mov %rdx, %rax
ret
```

- Unexpected Behavior
- Aggravate debugging difficulty

Solution: Test Configuration Exploration

- MCS: memorized configuration search
- Attached to test program generators

- Found 16 new bugs on GCC and LLVM
- Outperformed preexisting approaches
  - Detected ~2× more bugs
- Has been deployed in *Huawei* for testing their in-house compiler

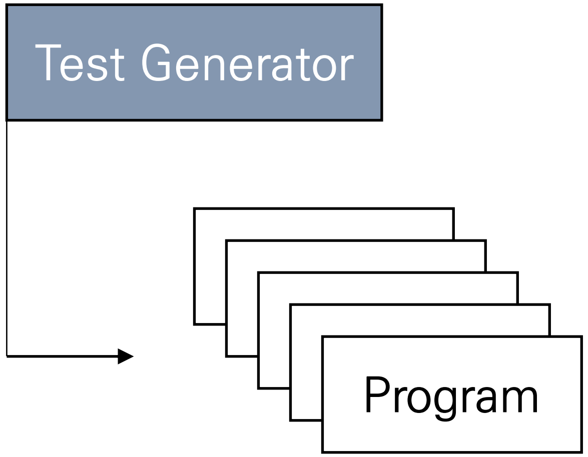
Ensuring Quality of Compilers

- Compiler testing
  - Test-program generators (e.g., Csmith [1])

[1] 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.

Ensuring Quality of Compilers

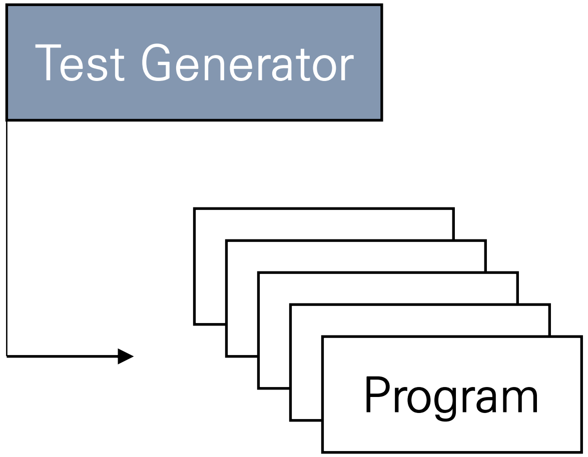
- Compiler testing
  - Test-program generators (e.g., Csmith [1])



[1] 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.

Ensuring Quality of Compilers

- Compiler testing
  - Test-program generators (e.g., Csmith [1])

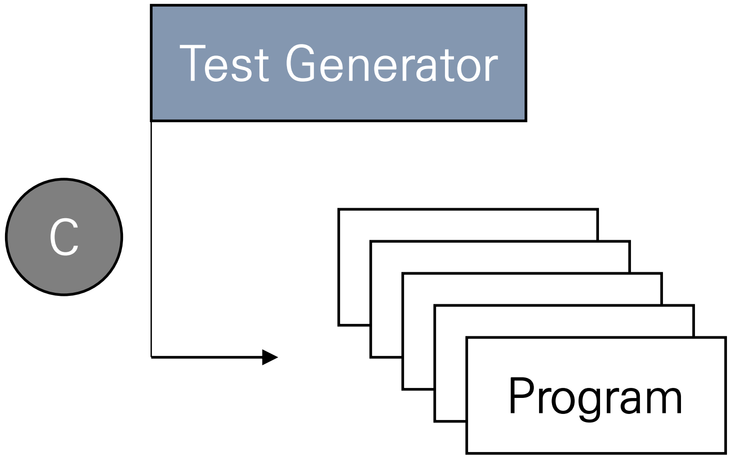



```
int add(int a, int b) {
    return a + b;
}
```

[1] 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.

Ensuring Quality of Compilers

- Compiler testing
  - Test-program generators (e.g., Csmith [1])
  - Test configuration

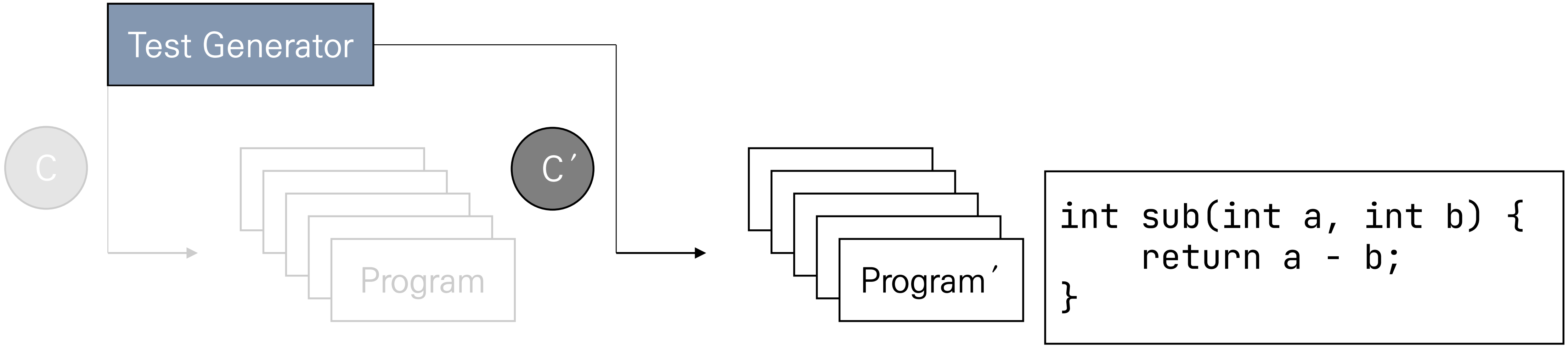



```
int add(int a, int b) {
    return a + b;
}
```

[1] 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.

Ensuring Quality of Compilers

- Compiler testing
  - Test-program generators (e.g., Csmith [1])
  - Test configuration



[1] 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.

Test Configuration Options

- Relying on only one (default) test configuration is not enough
  - Exploring test configuration [2]

[2] 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 Symposium on Software Testing and Analysis. 2012. pp. 78–88.

[3] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie, "Learning to prioritize test programs for compiler testing." in 2017 IEE/ACM 39th International Conference of Sortware Engineering., 2017. 99. 294–305.

Test Configuration Options

- Relying on only one (default) test configuration is not enough
  - Exploring test configuration [2]

- Simply varying test configuration is inefficient
  - Triggering compiler bugs tend to involve the specific combinations of some program features [3]

[2] 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 Symposium on Software Testing and Analysis. 2012. pp. 78-88.

[3] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie, "Learning to prioritize test programs for compiler testing." in 2017 IEE/ACM 39th International Conference of Sortware Engineering., 2017. 99. 294-305.

Test Configuration Options

- Relying on only one (default) test configuration is not enough
  - Exploring test configuration [2]

- Simply varying test configuration is inefficient
  - Triggering compiler bugs tend to involve the specific combinations of some program features [3]

Key observation

- **Options in configuration require elaborate coordination.**

[2] 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 Symposium on Software Testing and Analysis. 2012. pp. 78-88.

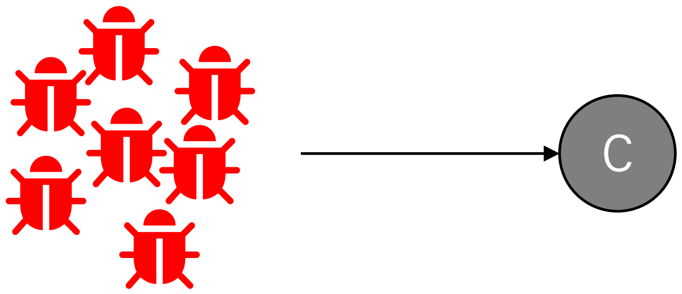
[3] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie, "Learning to prioritize test programs for compiler testing." in 2017 IEE/ACM 39th International Conference of Sortware Engineering., 2017. 99. 294-305.

MCS: Memoized Configuration Search

- *Offline* method: mining historical bugs to infer set of configurations

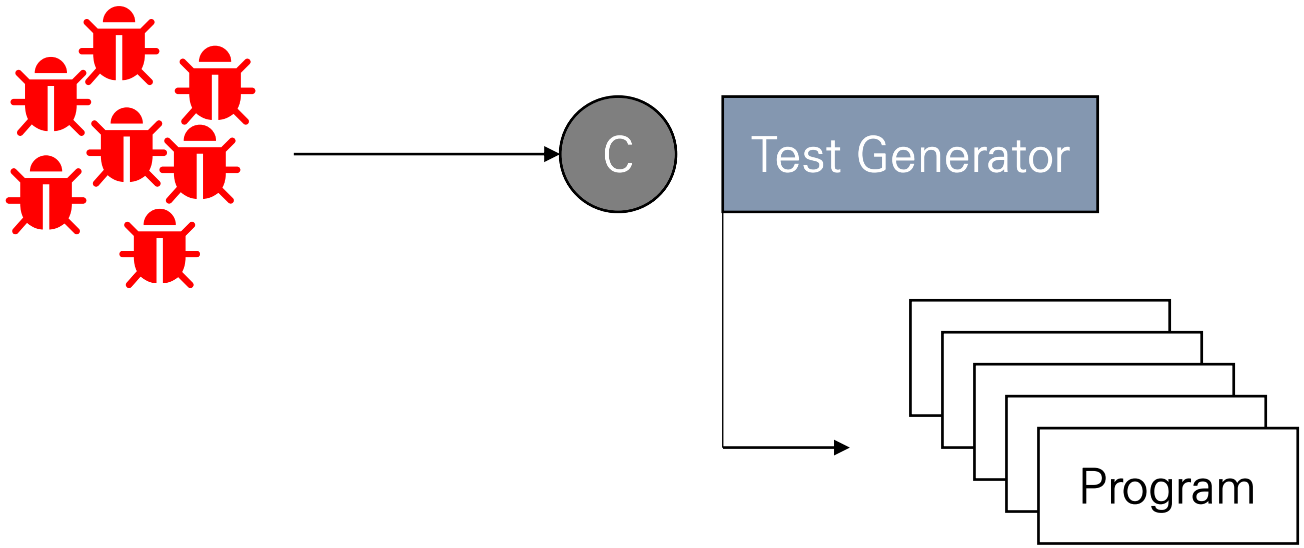
MCS: Memoized Configuration Search

- *Offline* method: mining historical bugs to infer set of configurations



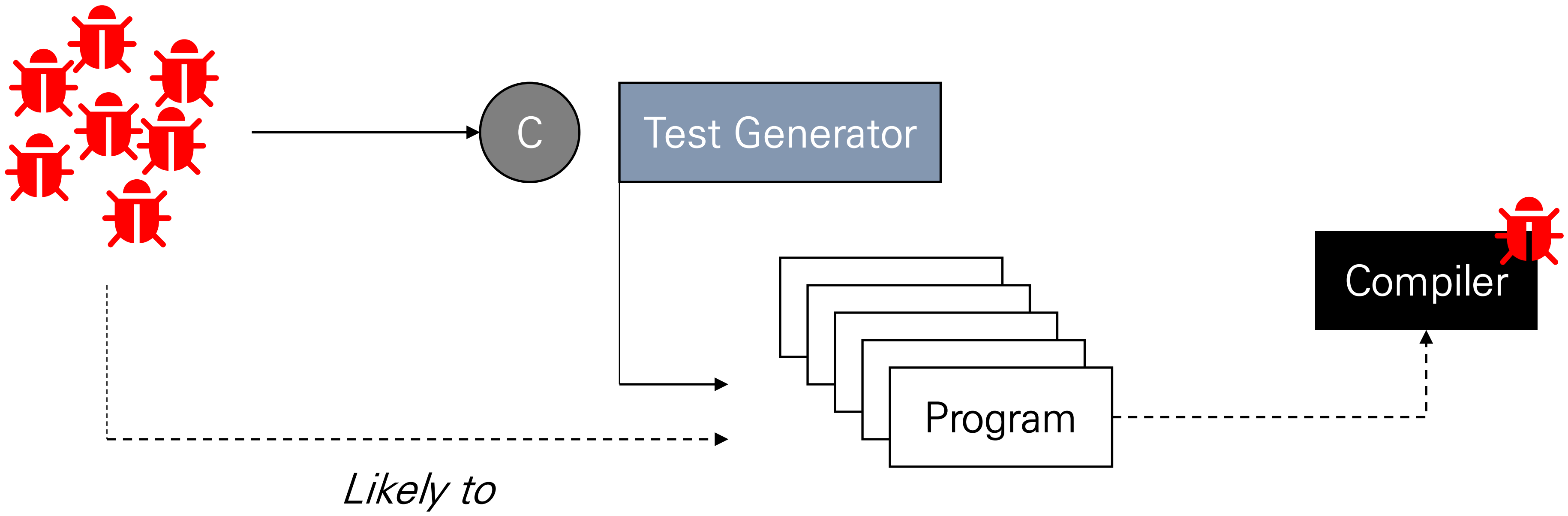
MCS: Memoized Configuration Search

- *Offline* method: mining historical bugs to infer set of configurations



MCS: Memoized Configuration Search

- *Offline* method: mining historical bugs to infer set of configurations



MCS: Memoized Configuration Search

- *Offline* method: mining historical bugs to infer set of configurations
- **Interleave** the process of searching for test configurations and the online testing process

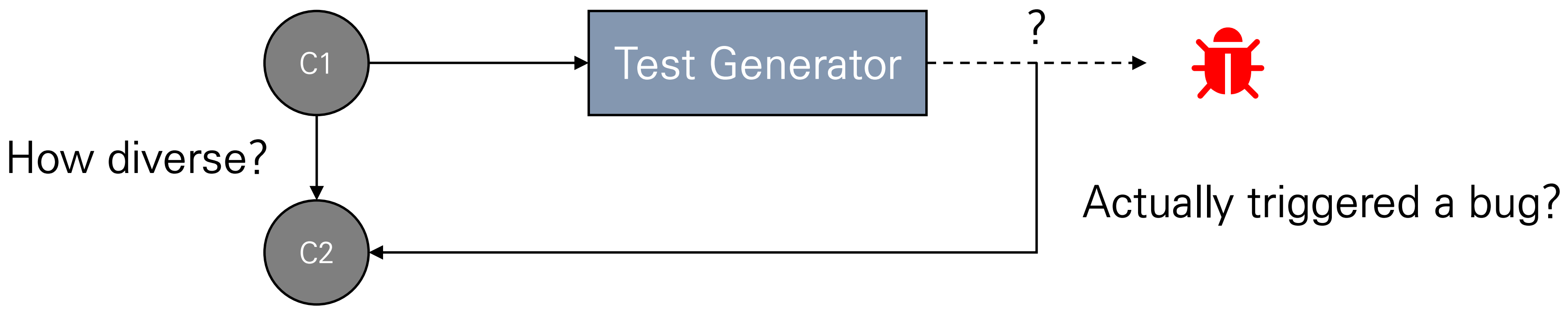
MCS: Memoized Configuration Search

- *Offline* method: mining historical bugs to infer set of configurations

- **Interleave** the process of searching for test configurations and the online testing process

MCS: Memoized Configuration Search

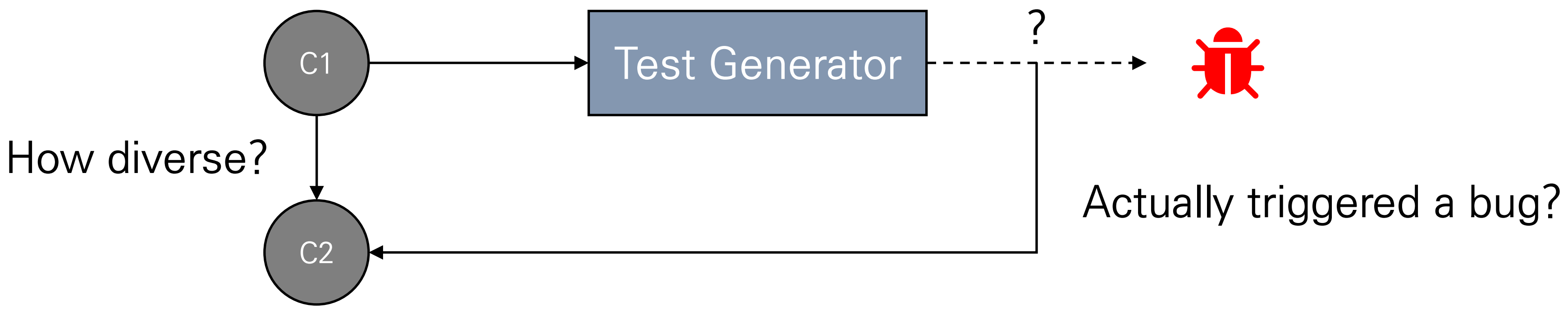
- *Offline* method: mining historical bugs to infer set of configurations
- **Interleave** the process of searching for test configurations and the online testing process



MCS: Memoized Configuration Search

- *Offline* method: mining historical bugs to infer set of configurations

- **Interleave** the process of searching for test configurations and the online testing process



Multi-agent Reinforcement Learning

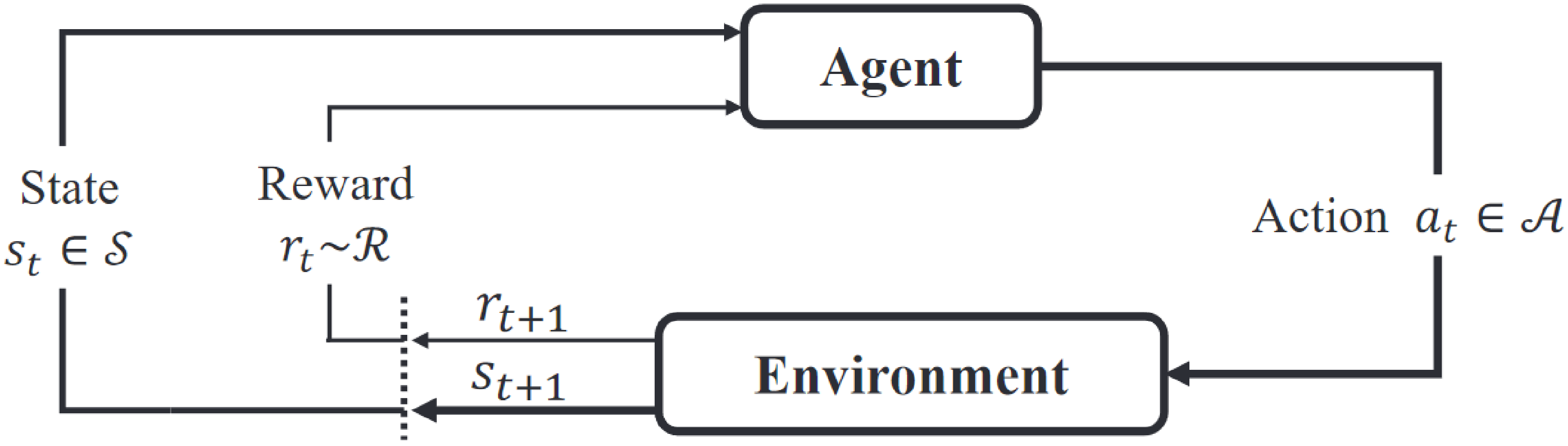
- "…. MCS innovatively adopt the *multi-agent RL* method to achieve our search goal. …"
- Wait, what is that?

Reinforcement Learning

- Agents learn an optimal control policy that interact with unknown environment
  - Choosing sequence that maximizes cumulative reward in a long run

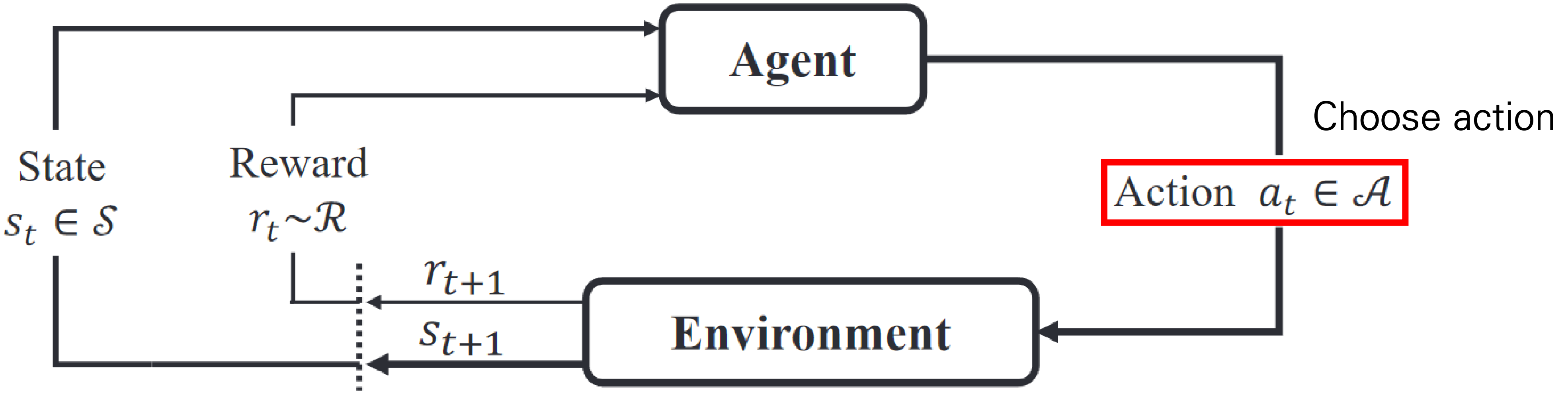
Reinforcement Learning

- Agents learn an optimal control policy that interact with unknown environment
  - Choosing sequence that maximizes cumulative reward in a long run



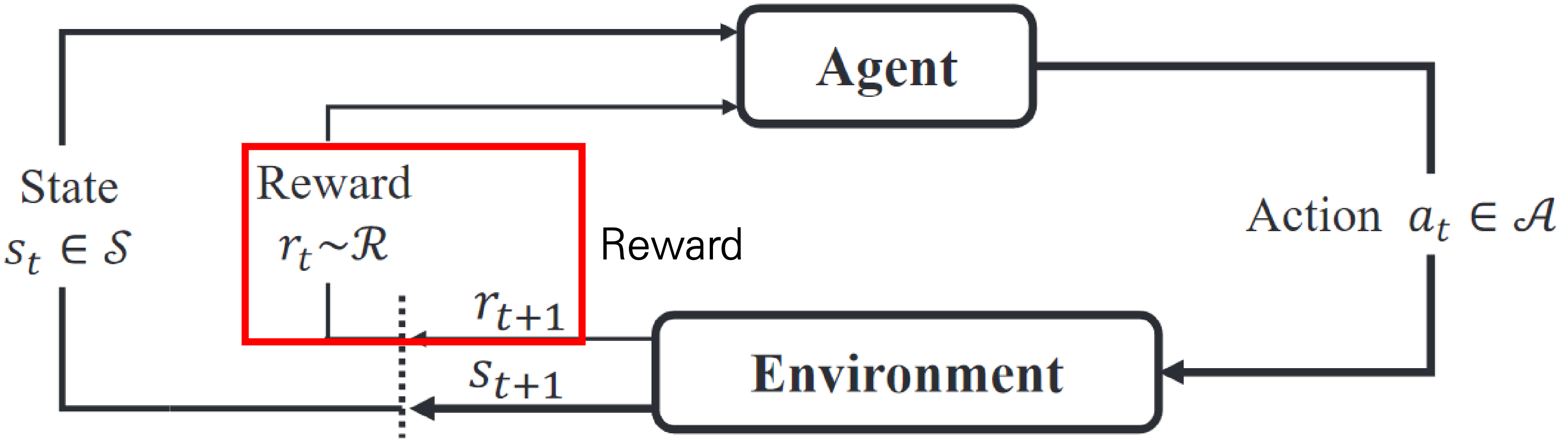
Reinforcement Learning

- Agents learn an optimal control policy that interact with unknown environment
  - Choosing sequence that maximizes cumulative reward in a long run



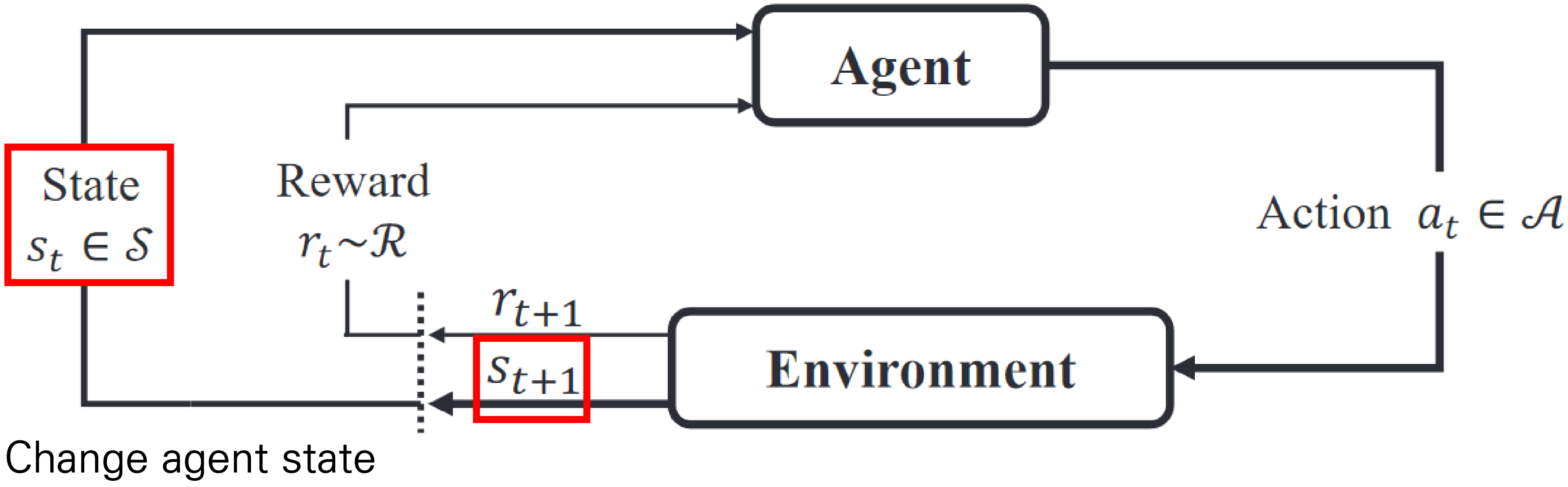
Reinforcement Learning

- Agents learn an optimal control policy that interact with unknown environment
  - Choosing sequence that maximizes cumulative reward in a long run



Reinforcement Learning

- Agents learn an optimal control policy that interact with unknown environment
  - Choosing sequence that maximizes cumulative reward in a long run



Reinforcement Learning

- Agents learn an optimal control policy that interact with unknown environment
  - Choosing sequence that maximizes cumulative reward in a long run

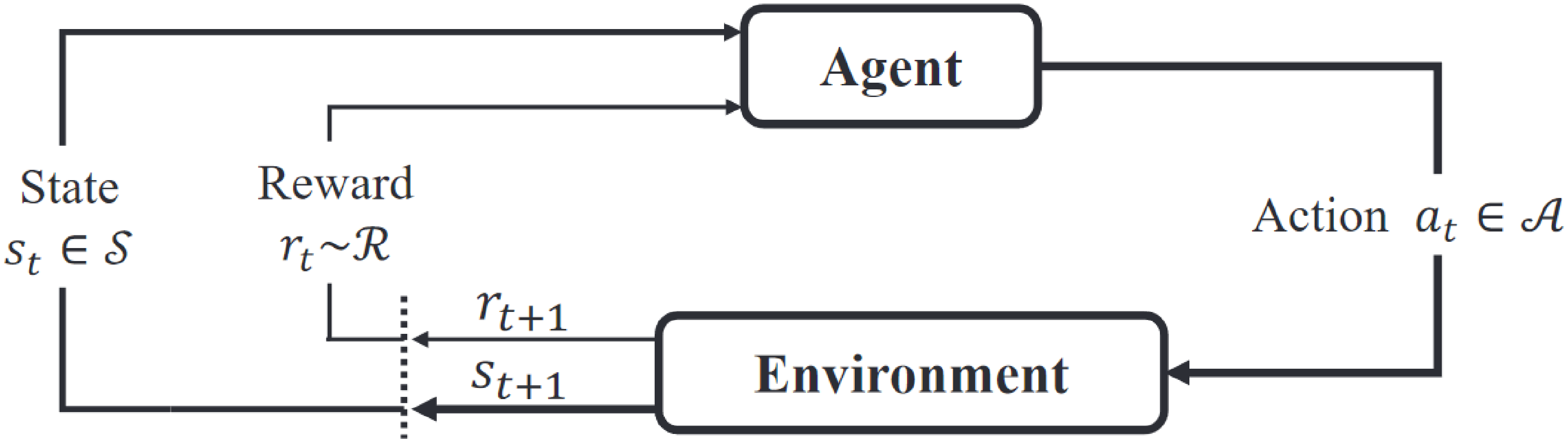
- Questions:
  - Why choose RL?
  - What is multi-agent RL and why choose multi-agent RL?

Reinforcement Learning

- Why choose RL?
  - It is suitable to the scenario.

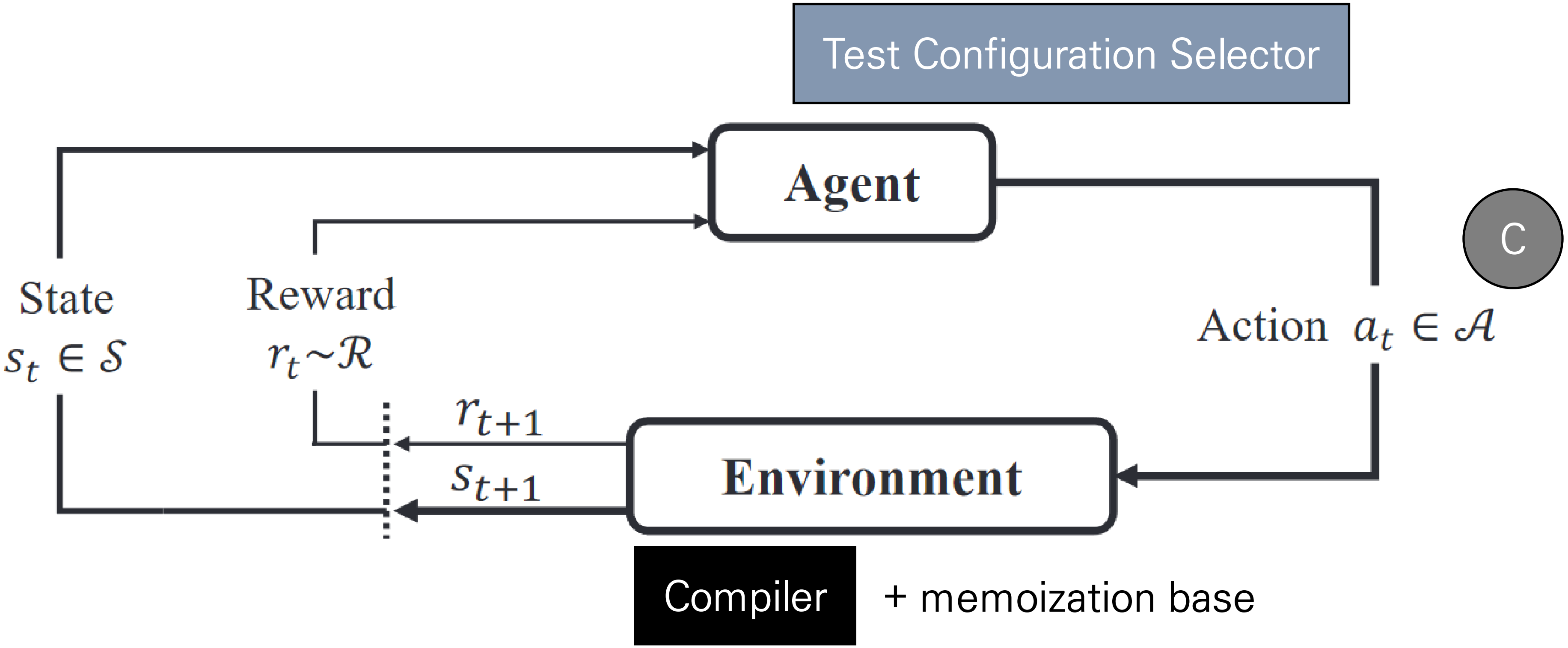
Reinforcement Learning

- Why choose RL?
  - It is suitable to the scenario.



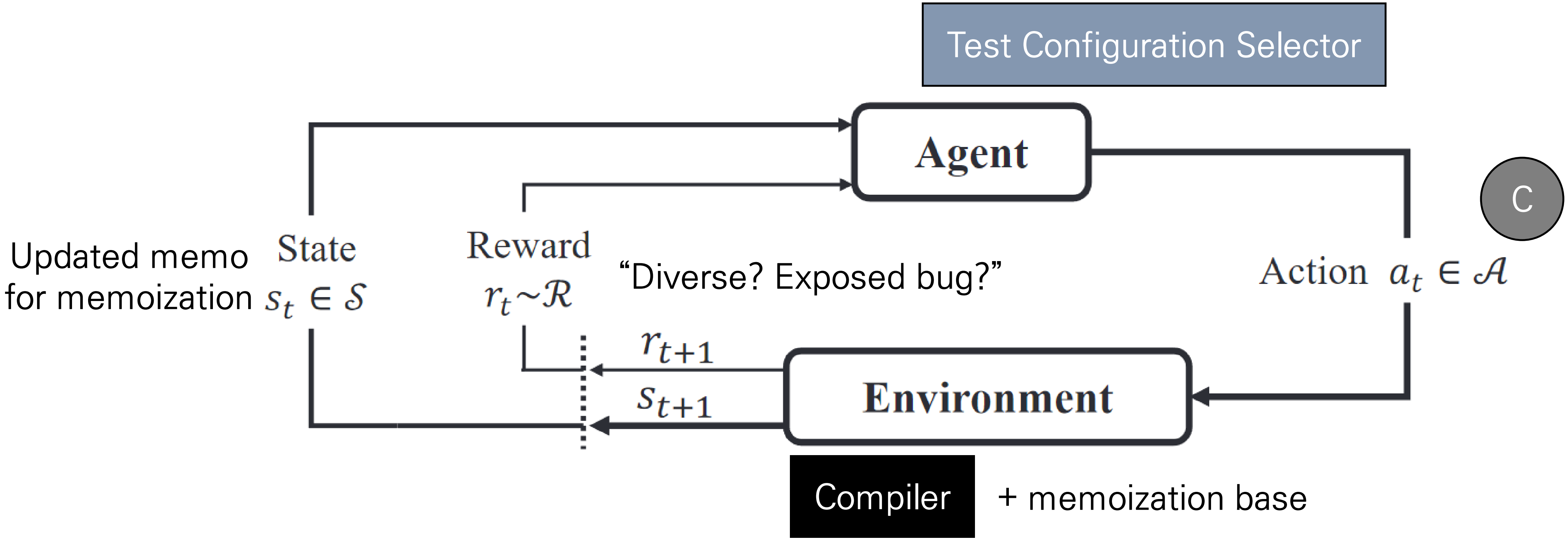
Reinforcement Learning

- Why choose RL?
  - It is suitable to the scenario.



Reinforcement Learning

- Why choose RL?
  - It is suitable to the scenario.



Reinforcement Learning

- Why choose RL?
  - It is suitable to the scenario.
  - It is effective for memorized search [4].

[4] 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.

[5] 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.

Reinforcement Learning

- Why choose RL?
  - It is suitable to the scenario.
  - It is effective for memorized search [4].

- Adopts A2C algorithm [5] for learning strategy
  - Skipped
  - Generate → *learn* → take action (update configuration) → loop

[4] 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.

[5] 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.

Multi-agent RL

- Triggering bug tends to involve the specific **combination** of several program features

Multi-agent RL

- Triggering bug tends to involve the specific **combination** of several program features
- MARL (**M**ulti-**A**gent **RL**)
  - Treat each configuration option as individual agent
  - Shares same environment

Multi-agent RL

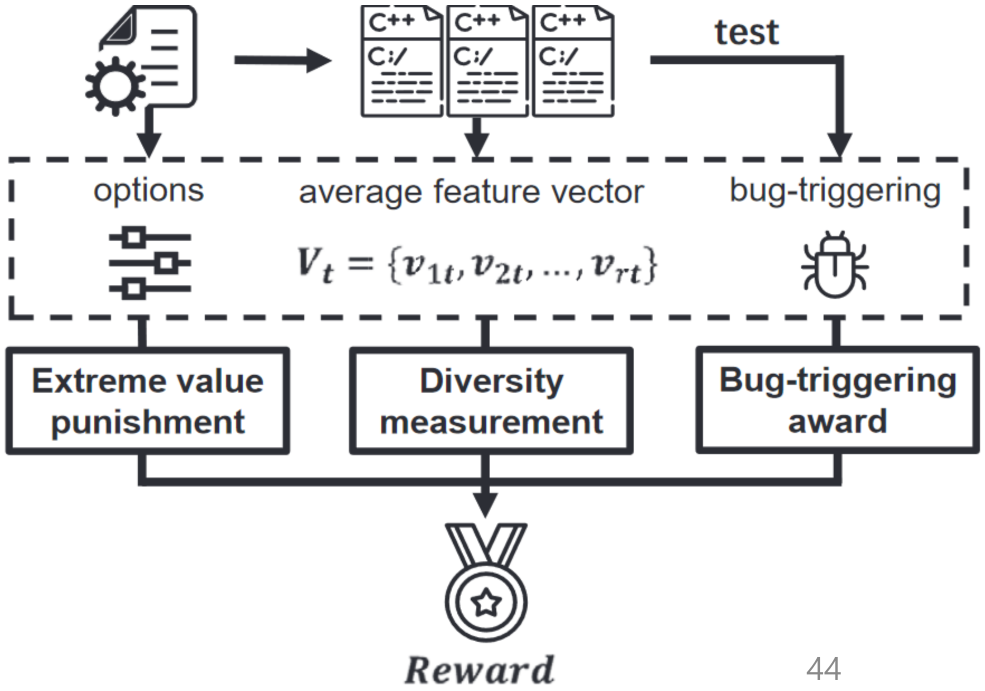
- Test configuration $c = \{o_1, \cdots, o_r\}$ (o_i: configuration option)
- Agent for option o_i: $\mathbf{agent}_i$
  - State of agent_i: $s_{it} = c_t = \{o_{1t}, \cdots, o_{rt}\}$
  - Action of agent_i: $a_{it} \in A_i$
- Learning task: for each agent_i, predict a action sequence $\langle a_{i1}, \cdots, a_{iT} \rangle$, which can produce the large cumulative rewards

Multi-agent RL

- MARL and the compiler configuration, in shorts:
  - Agents: each configuration option
  - Action: tuning (e.g., on/off)
  - Environment: compiler test campaign
  - Reward: “How diverse?” & “Exposed a bug?”
  - State: configuration set (option vector)

Approach

- Reward function: $R_t = R_t^{\text{div}} + R_t^{\text{trg}} + R_t^{\text{neg}}$
  - R_t^{div}: Diversity among configurations
  - R_t^{trg}: Testing results under the configuration
  - R_t^{neg}: Negative-effectiveness
    - Skipped
    - Avoid configuration of test time > 10 min.

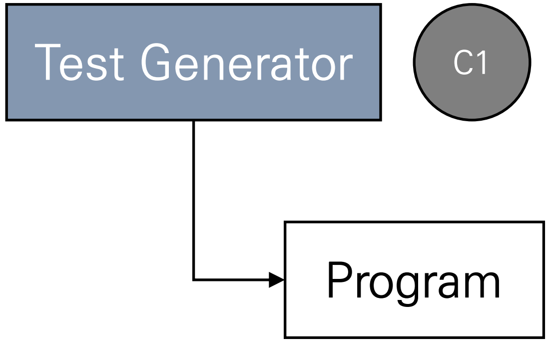


Diversity Component

- R_t^{div}: Diversity among configurations
- Extract feature vector of configuration
    - A test program includes language features based on the configuration

Diversity Component

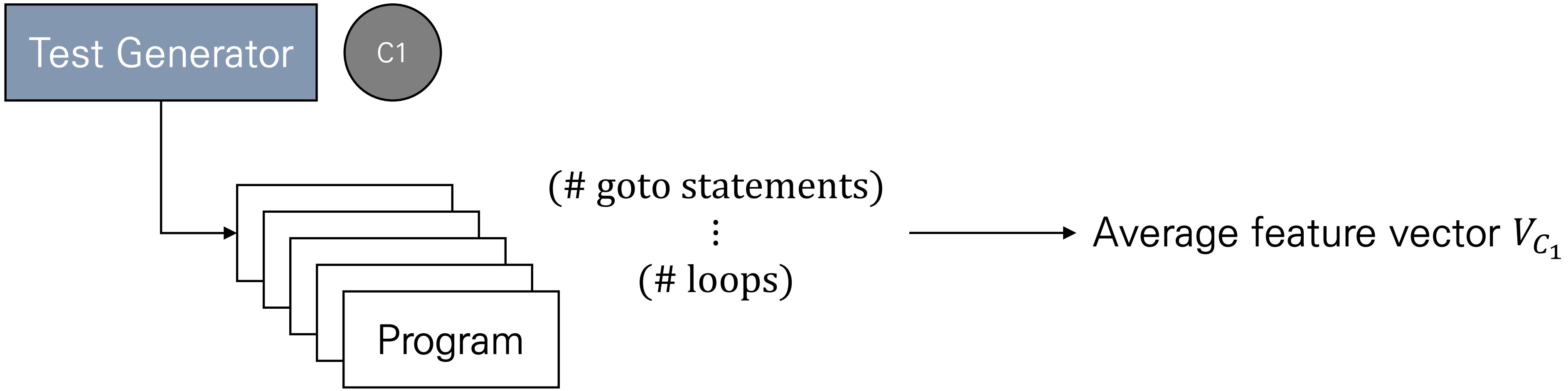
- R_t^{div}: Diversity among configurations
- Extract feature vector of configuration
  - A test program includes language features based on the configuration



$$\begin{pmatrix} \text{\# goto statements} \\ \vdots \\ \text{\# loops} \end{pmatrix}$$

Diversity Component

- R_t^{div}: Diversity among configurations
- Extract feature vector of configuration
  - A test program includes language features based on the configuration



Diversity Component

- R_t^{div}: Diversity among configurations
- Extract feature vector of configuration
  - A test program includes language features based on the configuration
  - Distance between two feature vectors: $\mathrm{Dist}(c_1, c_2) = 1 - \mathrm{c.s.}(V_1, V_2)$

Diversity Component

- R_t^{div}: Diversity among configurations
- Extract feature vector of configuration
  - A test program includes language features based on the configuration
  - Distance between two feature vectors: $\mathrm{Dist}(c_1, c_2) = 1 - \mathrm{c.}\,s.(V_1, V_2)$

Cosine similarity

Diversity Component

- R_t^{div}: Diversity among configurations
- Extract feature vector of configuration
  - A test program includes language features based on the configuration
  - Distance between two feature vectors: $\mathrm{Dist}(c_1, c_2) = 1 - \mathrm{c.s.}(V_1, V_2)$
- Diversity
  - C_h: set of explored test configurations that are closest to c_t
    (i.e., $\forall c_i \in C_h, c_j \in C_t, \mathrm{Dist}(c_i, c_t) \le \mathrm{Dist}\left(c_j, c_t\right)$)

Diversity Component

- R_t^{div}: Diversity among configurations
- Extract feature vector of configuration
  - A test program includes language features based on the configuration
  - Distance between two feature vectors: $\mathrm{Dist}(c_1, c_2) = 1 - \mathrm{c.s.}(V_1, V_2)$
- Diversity
  - C_h: set of explored test configurations that are closest to c_t
    (i.e., $\forall c_i \in C_h, c_j \in C_t, \mathrm{Dist}(c_i, c_t) \leq \mathrm{Dist}\left(c_j, c_t\right)$)

$$\mathrm{div}_t = \frac{1}{|C_h|} \sum_{c_i \in C_h} \mathrm{Dist}(c_i, c_t)$$

Diversity Component

- R_t^{div}: Diversity among configurations
- Extract feature vector of configuration
  - A test program includes language features based on the configuration
  - Distance between two feature vectors: $\text{Dist}(c_1, c_2) = 1 - \text{c.s.}(V_1, V_2)$
- Diversity
  - C_h: set of explored test configurations that are closest to c_t (i.e., $\forall c_i \in C_h, c_j \in C_t, \text{Dist}(c_i, c_t) \leq \text{Dist}(c_j, c_t)$)

$$\text{div}_t = \frac{1}{|C_h|} \sum_{c_i \in C_h} \text{Dist}(c_i, c_t) \qquad R_t^{\text{div}} = \frac{1}{m} \sum_{i=1}^{m} (\text{div}_t - \text{div}_{t-i})$$

Bug-triggering Award

- $R_t^{\mathbf{trg}}$: bug-triggering award in MCS

Bug-triggering Award

- $R_t^{\mathbf{trg}}$: bug-triggering award in MCS

$$R_t^{\mathbf{trg}} = \omega \times n_f$$

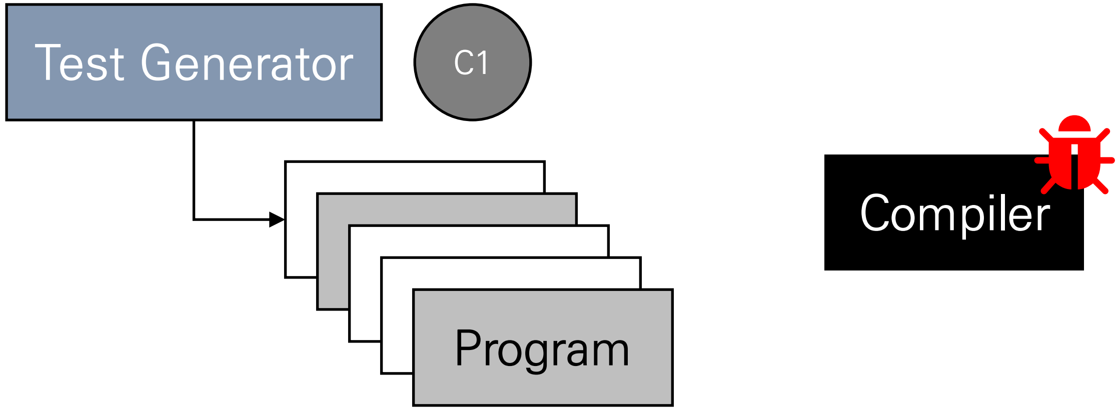
  - ω: constant coefficient
  - n_f: # of bug-triggering test programs generated under c_t

Bug-triggering Award

- R_t^{trg}: bug-triggering award in MCS

$$R_t^{\text{trg}} = \omega \times n_f$$

  - ω: constant coefficient
  - n_f: # of bug-triggering test programs generated under c_t

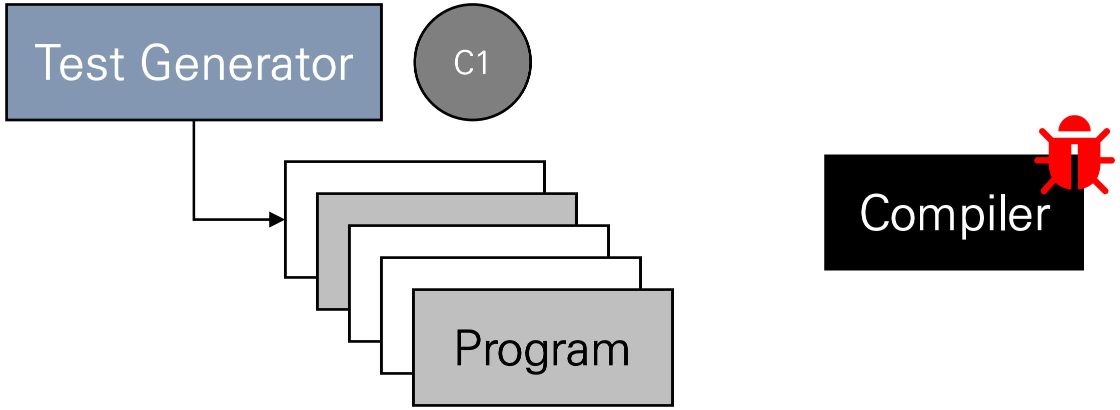


Bug−triggering Award

- R_t^{trg}: bug−triggering award in MCS

$$R_t^{\text{trg}} = \omega \times n_f$$

  - ω: constant coefficient
  - n_f: # of bug−triggering test programs generated under c_t



$R_1^{\text{trg}} = 2$

Experiment

- Research questions (RQ)
  - RQ1: how does MCS perform in detecting compiler bugs compared with SOTA approaches?
  - RQ2: How does MCS perform under different configurations?

Experiment

- Research questions (RQ)
  - RQ1: how does MCS perform in detecting compiler bugs compared with SOTA approaches?
  - RQ2: How does MCS perform under different configurations?

Configurations of MCS itself

Experiment

- Target compilers: GCC & LLVM

[1] 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.

[6] “Pytorch,” Accessed: 2021, https://pytorch.org.

[7] W. M. McKeeman, “Differential testing for software,” Digital Technical Journal, vol. 10, no. 1, pp. 100–107, 1998.

Experiment

- Target compilers: GCC & LLVM
- Test program generator: Csmith [1]

[1] 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.

[6] “Pytorch,” Accessed: 2021, https://pytorch.org.

[7] W. M. McKeeman, “Differential testing for software,” Digital Technical Journal, vol. 10, no. 1, pp. 100–107, 1998.

Experiment

- Target compilers: GCC & LLVM
- Test program generator: Csmith [1]
  - 71 options, generate 100 test programs under each configuration

[1] 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.

[6] “Pytorch,” Accessed: 2021, https://pytorch.org.

[7] W. M. McKeeman, “Differential testing for software,” Digital Technical Journal, vol. 10, no. 1, pp. 100–107, 1998.

Experiment

- Target compilers: GCC & LLVM
- Test program generator: Csmith [1]
  - 71 options, generate 100 test programs under each configuration
- Constants: $|C_h| = 10, m = 10, \omega = 4$

[1] 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.

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

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

Experiment

- Target compilers: GCC & LLVM
- Test program generator: Csmith [1]
  - 71 options, generate 100 test programs under each configuration
- Constants: $|C_h| = 10, m = 10, \omega = 4$
- Implementation: PyTorch [6]
- Test oracle: differential testing [7]

[1] 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.

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

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

Experiment

- Target compilers: GCC & LLVM
- Test program generator: Csmith [1]
  - 71 options, generate 100 test programs under each configuration
- Constants: $|C_h| = 10, m = 10, \omega = 4$
- Implementation: PyTorch [6]
- Test oracle: differential testing [7]

〉8 months experiment

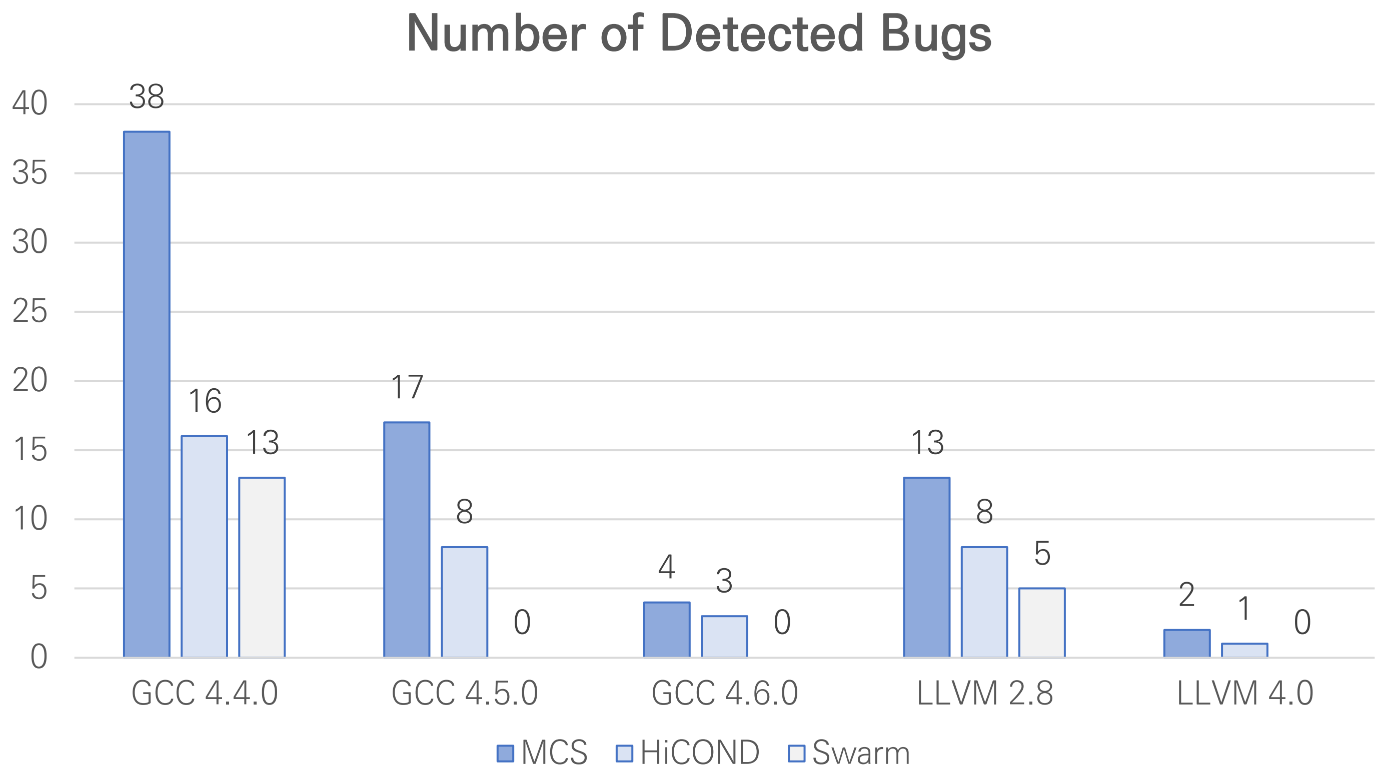
[1] X. Yang, Y.Chen., E. Eide, and J. Regehr. “Finding and understanding bugs in c compilers.” in Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, 2011. pp. 283–294.

[6] “Pytorch,” Accessed: 2021, https://pytorch.org.

[7] W. M. McKeeman, “Differential testing for software,” Digital Technical Journal, vol. 10, no. 1, pp. 100–107, 1998.

Experiment

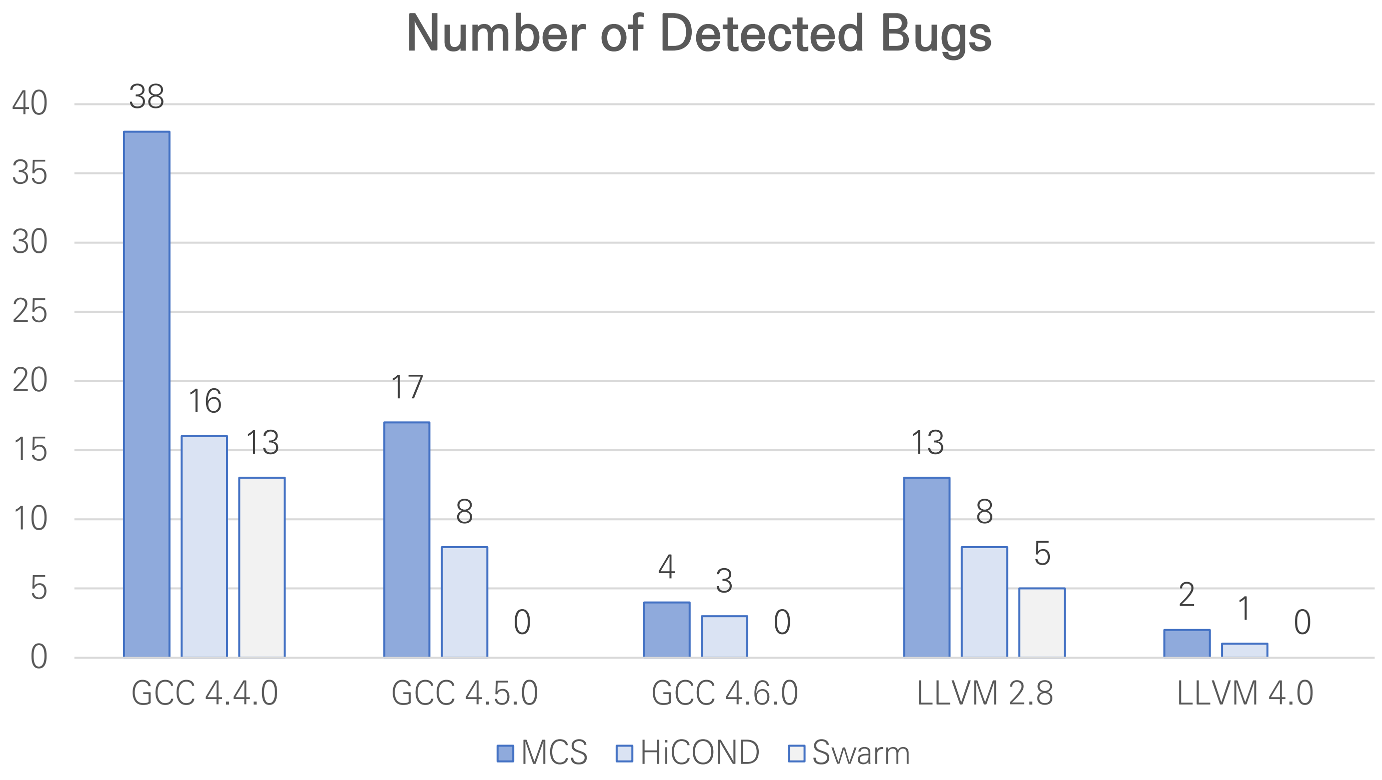
- RQ1: how does MCS perform in detecting compiler bugs compared with SOTA approaches?



* HiCOND: mines historical bugs to set configuration
* Swarm: randomly update test configuration

Experiment

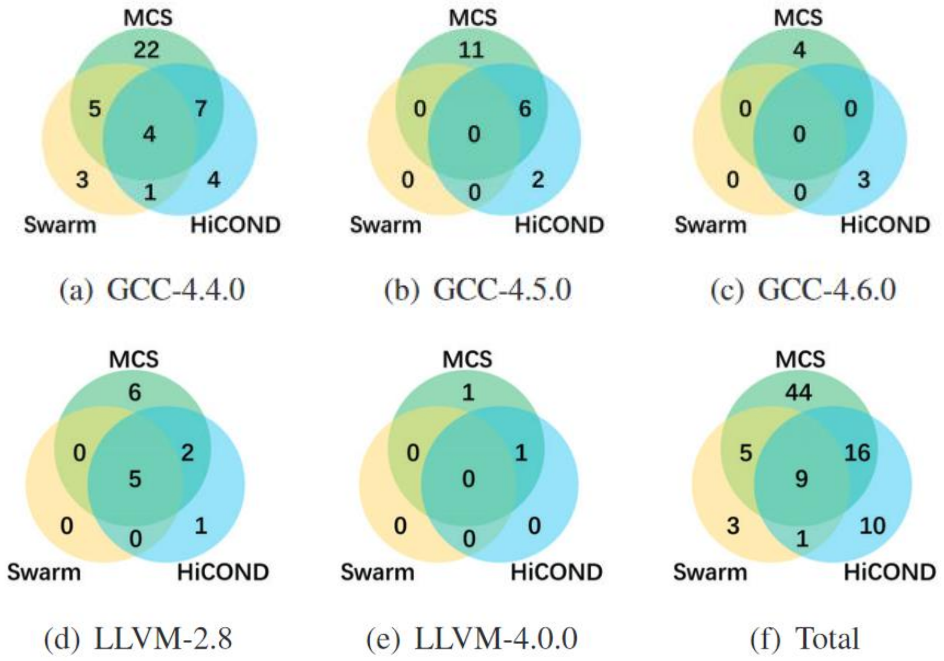
- RQ1: how does MCS perform in detecting compiler bugs compared with SOTA approaches?



* HiCOND: mines historical bugs to set configuration
* Swarm: randomly update test configuration

Experiment

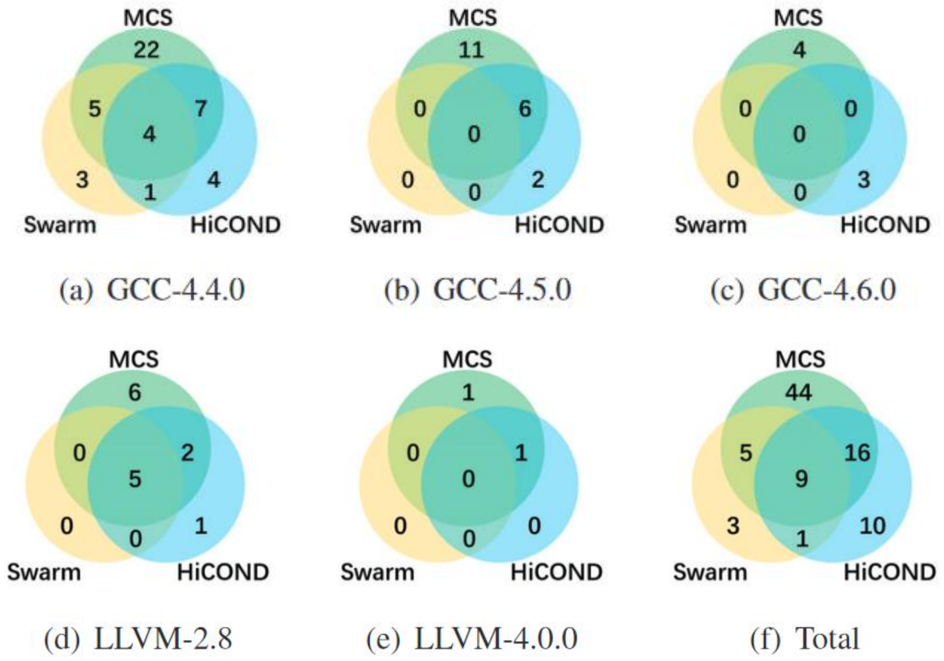
- RQ1: how does MCS perform in detecting compiler bugs compared with SOTA approaches?



* HiCOND: mines historical bugs to set configuration
* Swarm: randomly update test configuration

Experiment

- RQ1: how does MCS perform in detecting compiler bugs compared with SOTA approaches?
  - HiCOND:
    focus on a certain portion of input space
  - Swarm:
    no guidance,
    can explore input space missed by other approaches

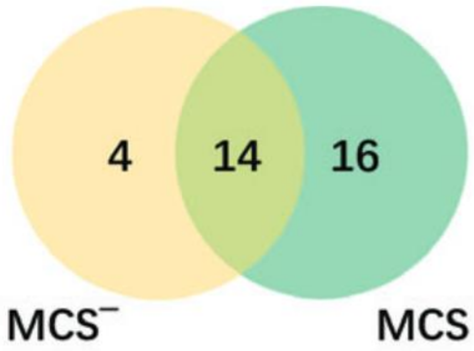


Experiment

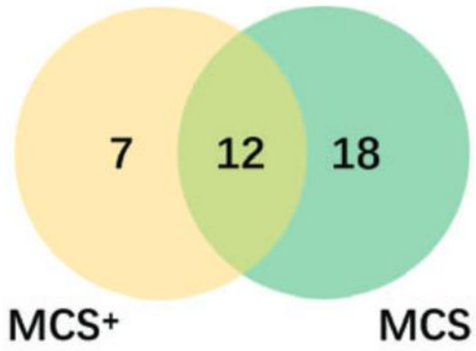
- RQ2: How does MCS perform under different configurations?
  - Removing $R_t^{\mathbf{trg}}$, $R_t^{\mathbf{Neg}}$ terms in reward function

Experiment

- RQ2: How does MCS perform under different configurations?
  - Removing R_t^{trg}, R_t^{Neg} terms in reward function



(a) w/o extreme value punishment.



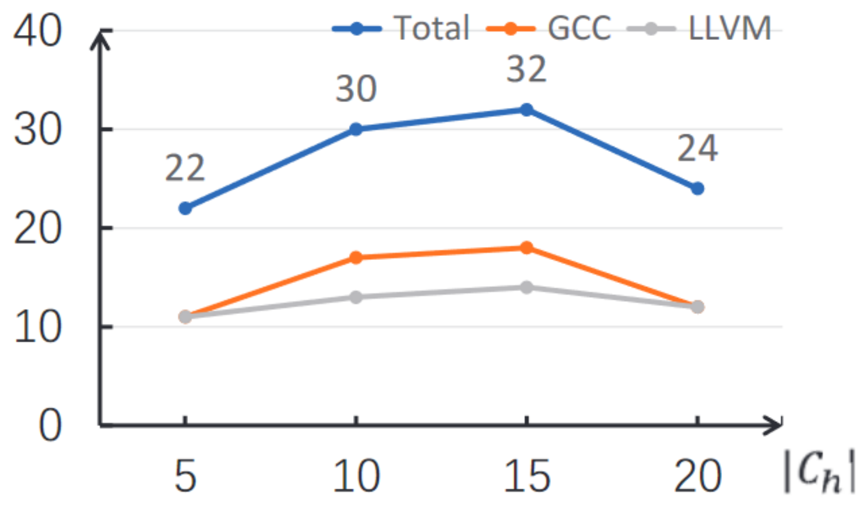
(b) w/o bug-triggering award.

Experiment

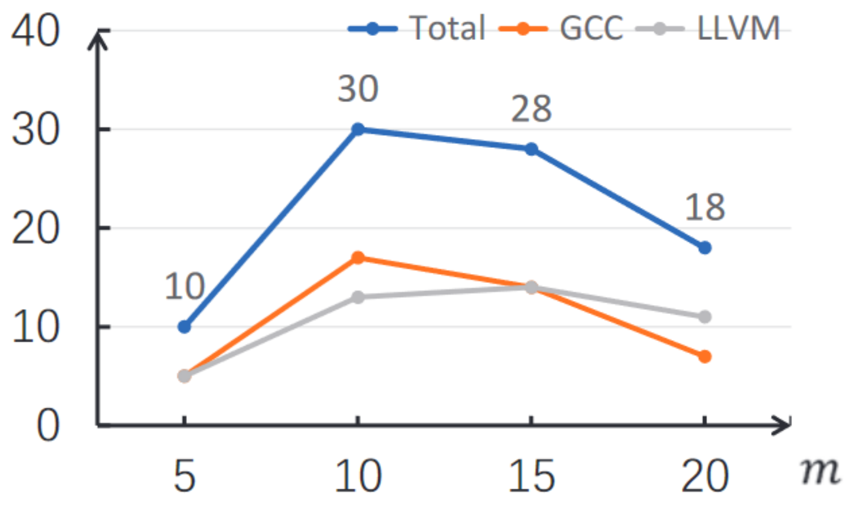
- RQ2: How does MCS perform under different configurations?
  - Removing $R_t^{\mathbf{trg}}$, $R_t^{\mathbf{Neg}}$ terms in reward function
  - Modifying $|C_h| \in \{5, 10, 15, 20\}$, $m = \{5, 10, 15, 20\}$

Experiment

- RQ2: How does MCS perform under different configurations?
  - Removing R_t^{trg}, R_t^{Neg} terms in reward function
  - Modifying $|C_h| \in \{5, 10, 15, 20\}$, $m = \{5, 10, 15, 20\}$



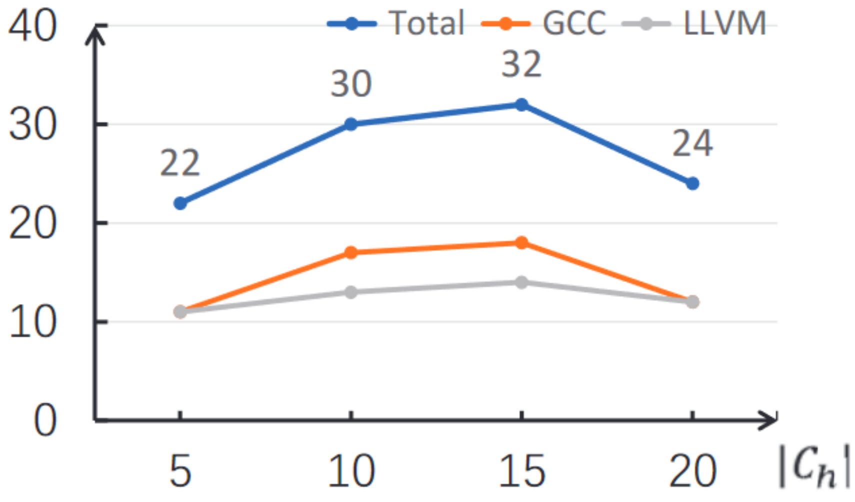
($m = 10$)



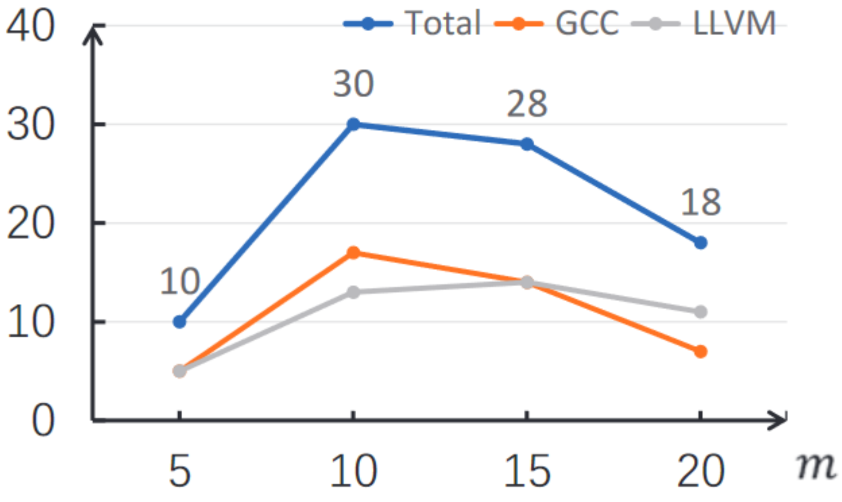
($|C_h| = 10$)

Experiment

- RQ2: How does MCS perform under different configurations?
  - Removing R_t^{trg}, R_t^{Neg} terms in reward function
  - Modifying $|C_h| \in \{5, 10, 15, 20\}$, $m = \{5, 10, 15, 20\}$
    - Too small/large parameters lowers performance (efficiency, search space)



$(m = 10)$



$(|C_h| = 10)$

My Review

- Key observation is effectively connected to interesting idea
  - **Coordination of option → MLIR**
  - Sufficiently supported by references

- Policy is gradually learned
  - Ineffectiveness of config generation in initial stage of RL
  - How about trashing some initially-generated configurations after some steps?

- Can there be other terms for reward function?

Summary

- MCS: Memoized Configuration Search
  - Via MARL
  - Reward function: triggering bugs, diversity, extreme value penalty

- MCS largely outperforms the SOTA approaches

- Fresh contributions
  - Found new 16 GCC and LLVM bugs
  - Has been deployed in *Huawei* for testing their in-house compiler

Aux: RL and our purpose

- 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.

Aux: dead code

- 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. Then, MCS calculates diversity based on the feature vectors.

Aux: negative effectiveness

- 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^neg_t = λ, otherwise 0.

Aux: using C_h

- 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 ct and a set of closest explored ones (rather than all the explored ones) as the approximation.

Aux: using m

- 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.