*The 46th International Conference on Software Engineering (ICSE 2024)*

# Beyond a Joke: Dead Code Elimination Can Delete Live Code

**Haoxin Tu**, Lingxiao Jiang, Debin Gao (Singapore Management University)
He Jiang (Dalian University of Technology)

18/04/2024, Lisbon

# Background: Dead Code Elimination



- **What is Dead Code Elimination (DCE)?**
  - A fundamental compiler optimization technique that removes dead code (e.g., unreachable or reachable but whose results are unused) in the program

# Background: Dead Code Elimination

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

## ❑ What is Dead Code Elimination (DCE)?

- A fundamental compiler optimization technique that removes dead code (e.g., unreachable or reachable but whose results are unused) in the program

```c
int foo(void) {
  int a = 24;
  int b = 25; /* Assignment to dead variable */
  int c;
  c = a * 4;
  return c;
  b = 24; /* Unreachable code */
  return 0;
}
```

```c
int main(void) {
  int a = 5, b = 6, c = 0;
  c = a * (b / 2);
  if (0) {   /* DEBUG */
    int ret = foo();
    printf("%d\n", ret);
  }
  return c;
}
```

Examples are cited from https://en.wikipedia.org/wiki/Dead-code_elimination#Examples

# Background: Dead Code Elimination

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

## ❑ What is Dead Code Elimination (DCE)?

- A fundamental compiler optimization technique that removes dead code (e.g., unreachable or reachable but whose results are unused) in the program

```c
int foo(void) {
  int a = 24;
  int b = 25; /* Assignment to dead variable */
  int c;
  c = a * 4;
  return c;
  b = 24; /* Unreachable code */
  return 0;
}
```

```c
int main(void) {
  int a = 5, b = 6, c = 0;
  c = a * (b / 2);
  if (0) {   /* DEBUG */
    int ret = foo();
    printf("%d\n", ret);
  }
  return c;
}
```

Examples are cited from https://en.wikipedia.org/wiki/Dead-code_elimination#Examples

# Background: Dead Code Elimination

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

## ❑ What is Dead Code Elimination (DCE)?

- A fundamental compiler optimization technique that removes dead code (e.g., unreachable or reachable but whose results are unused) in the program

```c
int foo(void) {
  int a = 24;
  int b = 25; /* Assignment to dead variable */
  int c;
  c = a * 4;
  return c;
  b = 24; /* Unreachable code */
  return 0;
}
```

```c
int main(void) {
  int a = 5, b = 6, c = 0;
  c = a * (b / 2);
  if (0) {   /* DEBUG */
    int ret = foo();
    printf("%d\n", ret);
  }
  return c;
}
```

Examples are cited from https://en.wikipedia.org/wiki/Dead-code_elimination#Examples

# Background: Dead Code Elimination

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

## ❑ What is Dead Code Elimination (DCE)?

- A fundamental compiler optimization technique that removes dead code (e.g., unreachable or reachable but whose results are unused) in the program

```c
int foo(void) {
  int a = 24;
  int b = 25; /* Assignment to dead variable */
  int c;
  c = a * 4;
  return c;
  b = 24; /* Unreachable code */
  return 0;
}
```

```c
int main(void) {
  int a = 5, b = 6, c = 0;
  c = a * (b / 2);
  if (0) {   /* DEBUG */
    int ret = foo();
    printf("%d\n", ret);
  }
  return c;
}
```

- Benefits of DCE: produce ***smaller*** or ***faster*** executables
  - Many other applications and languages (Java, Go, and Rust, etc.)

Examples are cited from https://en.wikipedia.org/wiki/Dead-code_elimination#Examples

# Motivation

# Motivation



**Question: Can DCE happen to erroneously delete live code?**

# Motivation

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

**Question: Can DCE happen to erroneously delete live code?**

- **Motivating example**

```
int idx = 0;
int a = 0;
void __attribute__((noinline)) marker_2() { ++idx; }
static void c() { marker_2(); }
void d(int j) { for (;;) ; } // infinite loop
void e() { for(int i = 0; i < 100; m++) d(i);}
void f() {
  e(); // live code here is erroneously deleted
  c();
}
void g() {if (a == 0x99) f();}

int main(int argc, char* argv[]) {
  a = strtol(argv[1], NULL, 16); // when a = 0x99, the bug triggers
  g();
  printf("%d", idx);
  return 0;
} /* affecting -O1 and above on LLVM-11.0.1 downward versions */
```

**Figure 1: A *miscompilation* bug[2] detected by XDEAD**

# Motivation

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

**Question: Can DCE happen to erroneously delete live code?**

- **Motivating example**

```
int idx = 0;
int a = 0;
void __attribute__((noinline)) marker_2() { ++idx; }
static void c() { marker_2(); }
void d(int j) { for (;;) ; } // infinite loop
void e() { for(int i = 0; i < 100; m++) d(i);}
void f() {
  e(); // live code here is erroneously deleted
  c();
}
void g() {if (a == 0x99) f();}

int main(int argc, char* argv[]) {
  a = strtol(argv[1], NULL, 16); // when a = 0x99, the bug triggers
  g();
  printf("%d", idx);
  return 0;
} /* affecting -O1 and above on LLVM-11.0.1 downward versions */
```

**Figure 1: A *miscompilation* bug[2] detected by XDEAD**

# Motivation

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

**Question: Can DCE happen to erroneously delete live code?**

- **Motivating example**

```c
int idx = 0;
int a = 0;
void __attribute__((noinline)) marker_2() { ++idx; }
static void c() { marker_2(); }
void d(int j) { for (;;) ; } // infinite loop
void e() { for(int i = 0; i < 100; m++) d(i);}
void f() {
  e(); // live code here is erroneously deleted
  c();
}
void g() {if (a == 0x99) f();}

int main(int argc, char* argv[]) {
  a = strtol(argv[1], NULL, 16); // when a = 0x99, the bug triggers
  g();
  printf("%d", idx);
  return 0;
} /* affecting -O1 and above on LLVM-11.0.1 downward versions */
```

**Figure 1: A *miscompilation* bug[2] detected by XDEAD**

Executor x86-64 clang 12.0.0 (C, Editor #1)

A Wrap lines

x86-64 clang 12.0.0 | -O3 -std=c99

(Input) 0x99

```
Program returned: 143
Program stderr
```

(Output)

```
Killed - processing time exceeded
Program terminated with signal: SIGKILL
```

**Executable 1**

https://godbolt.org/z/z7zxexfr1

# Motivation

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

**Question: Can DCE happen to erroneously delete live code?**

- **Motivating example**

```
int idx = 0;
int a = 0;
void __attribute__((noinline)) marker_2() { ++idx; }
static void c() { marker_2(); }
void d(int j) { for (;;) ; } // infinite loop
void e() { for(int i = 0; i < 100; m++) d(i);}
void f() {
  e(); // live code here is erroneously deleted
  c();
}
void g() {if (a == 0x99) f();}

int main(int argc, char* argv[]) {
  a = strtol(argv[1], NULL, 16); // when a = 0x99, the bug triggers
  g();
  printf("%d", idx);
  return 0;
} /* affecting -O1 and above on LLVM-11.0.1 downward versions */
```

**Figure 1: A *miscompilation* bug[2] detected by XDEAD**

Executor x86-64 clang 12.0.0 (C, Editor #1)

Wrap lines

x86-64 clang 12.0.0 | -O3 -std=c99

(Input) 0x99

```
Program returned: 143
Program stderr

Killed - processing time exceeded
Program terminated with signal: SIGKILL
```

(Output)

**Executable 1**

Executor x86-64 clang 11.0.1 (C, Editor #1)

Wrap lines

x86-64 clang 11.0.1 | -O3 -std=c99

(Input) 0x99

```
Program returned: 0
Program stdout
1
```

(Output)

**Wrong binary code** **Executable 2**

https://godbolt.org/z/z7zxexfr1

# Motivation

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

**Question: Can DCE happen to erroneously delete live code?**

- **Motivating example**

```
1  int idx = 0;
2  int a = 0;
3  void __attribute__((noinline)) marker_2() { ++idx; }
4  static void c() { marker_2(); }
5  void d(int j) { for (;;) ; } // infinite loop
6  void e() { for(int i = 0; i < 100; m++) d(i);}
7  void f() {
8    e(); // live code here is erroneously deleted
9    c();
10 }
11 void g() {if (a == 0x99) f();}
12
13 int main(int argc, char* argv[]) {
14   a = strtol(argv[1], NULL, 16); // when a = 0x99, the bug triggers
15   g();
16   printf("%d", idx);
17   return 0;
18 } /* affecting -O1 and above on LLVM-11.0.1 downward versions */
```

**Figure 1: A *miscompilation* bug[2] detected by XDEAD**

(Input) 0x99

(Output)

```
Executor x86-64 clang 12.0.0 (C, Editor #1)
Wrap lines
x86-64 clang 12.0.0 | -O3 -std=c99
0x99
Program returned: 143
Program stderr
Killed - processing time exceeded
Program terminated with signal: SIGKILL
```

**Executable 1**

(Input) 0x99

(Output)

```
Executor x86-64 clang 11.0.1 (C, Editor #1)
Wrap lines
x86-64 clang 11.0.1 | -O3 -std=c99
0x99
Program returned: 0
Program stdout
1
```

**Wrong binary code** **Executable 2**

https://godbolt.org/z/z7zxexfr1

# Solution: Xdead

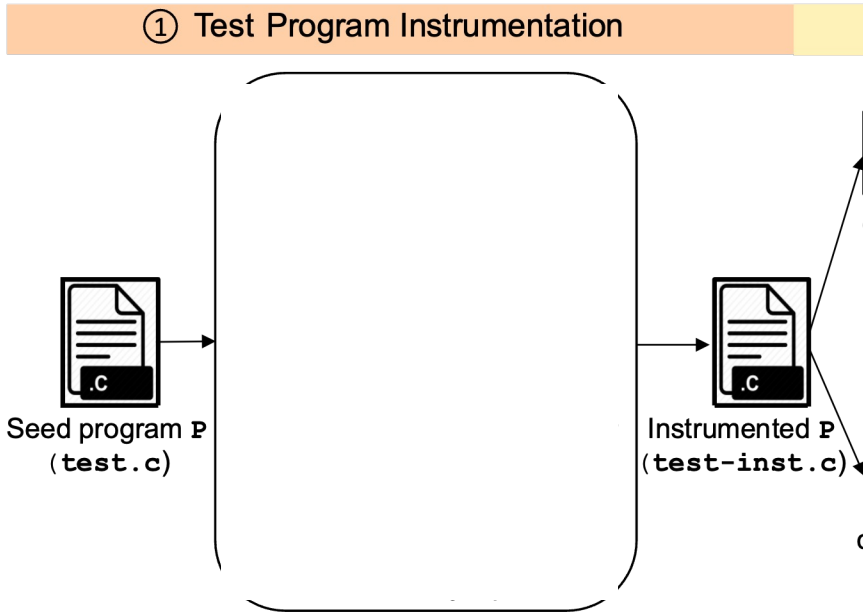
# Solution: Xdead



- **Key insights: divergence indicates DCE bugs**
  - Identify the divergent portions in binary first
  - leverage symbolic execution to reveal the divergent portion

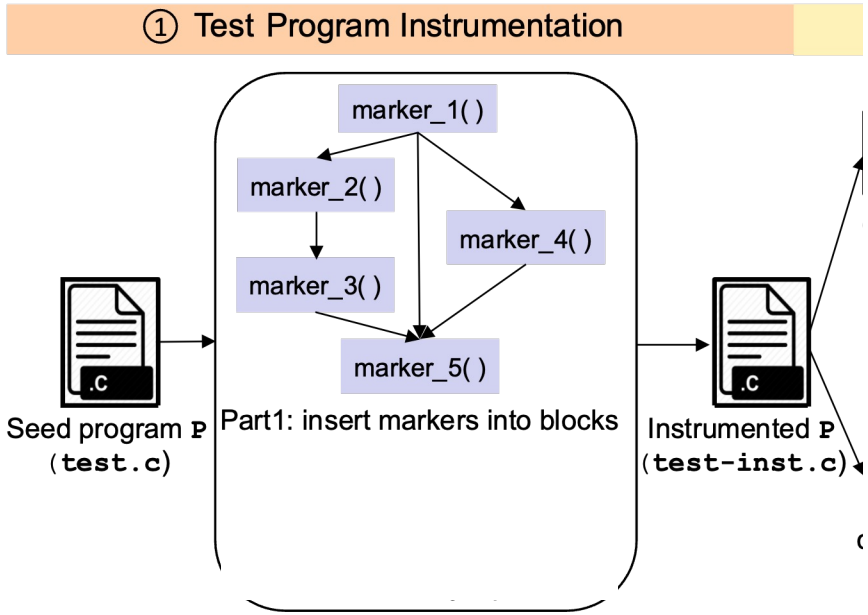
# Solution: Xdead

- **Key insights: divergence indicates DCE bugs**
  - Identify the divergent portions in binary first
  - leverage symbolic execution to reveal the divergent portion



# Solution: Xdead

- **Key insights: divergence indicates DCE bugs**
  - Identify the divergent portions in binary first
  - leverage symbolic execution to reveal the divergent portion

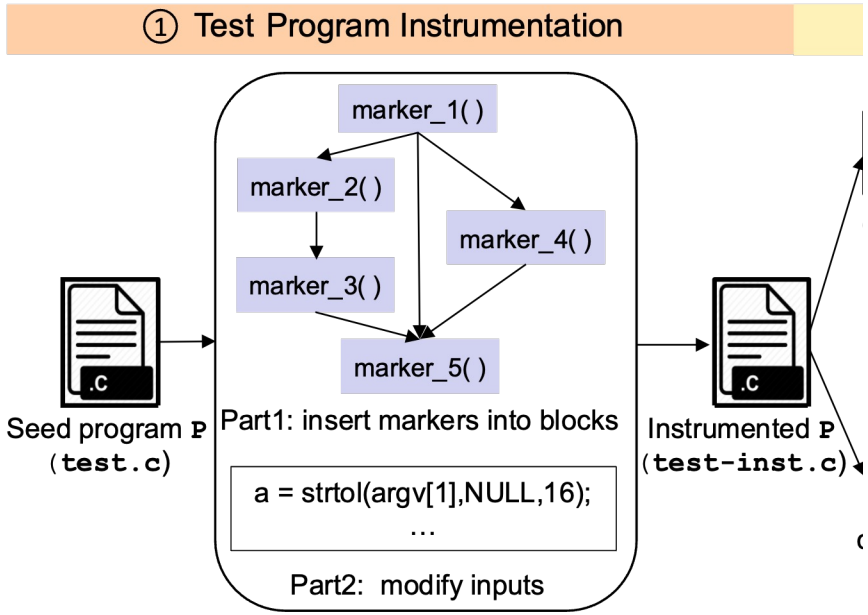



**The marker function:**

```
void __attribute__((noinline)) marker_i() { ++idx; };
```

# Solution: Xdead

- **Key insights: divergence indicates DCE bugs**
  - Identify the divergent portions in binary first
  - leverage symbolic execution to reveal the divergent portion

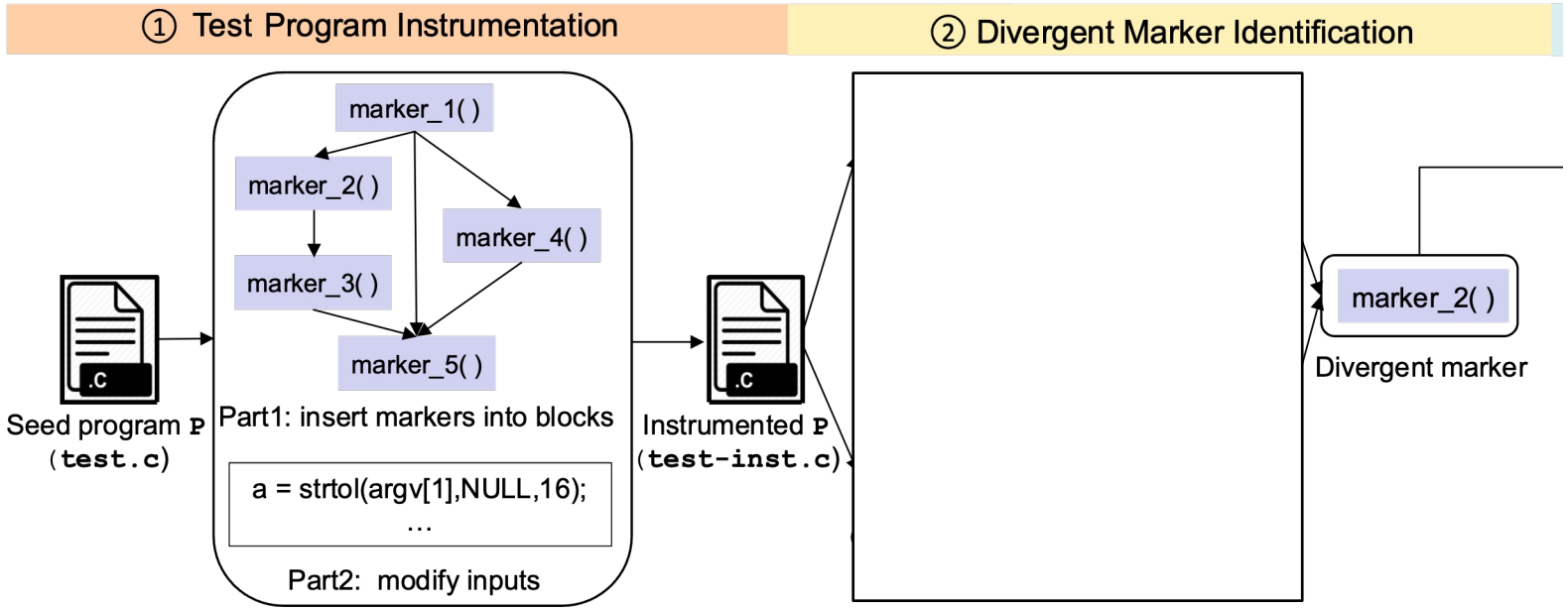



**The marker function:**

```
void __attribute__((noinline)) marker_i() { ++idx; };
```

# Solution: Xdead

- **Key insights: divergence indicates DCE bugs**
  - Identify the divergent portions in binary first
  - leverage symbolic execution to reveal the divergent portion

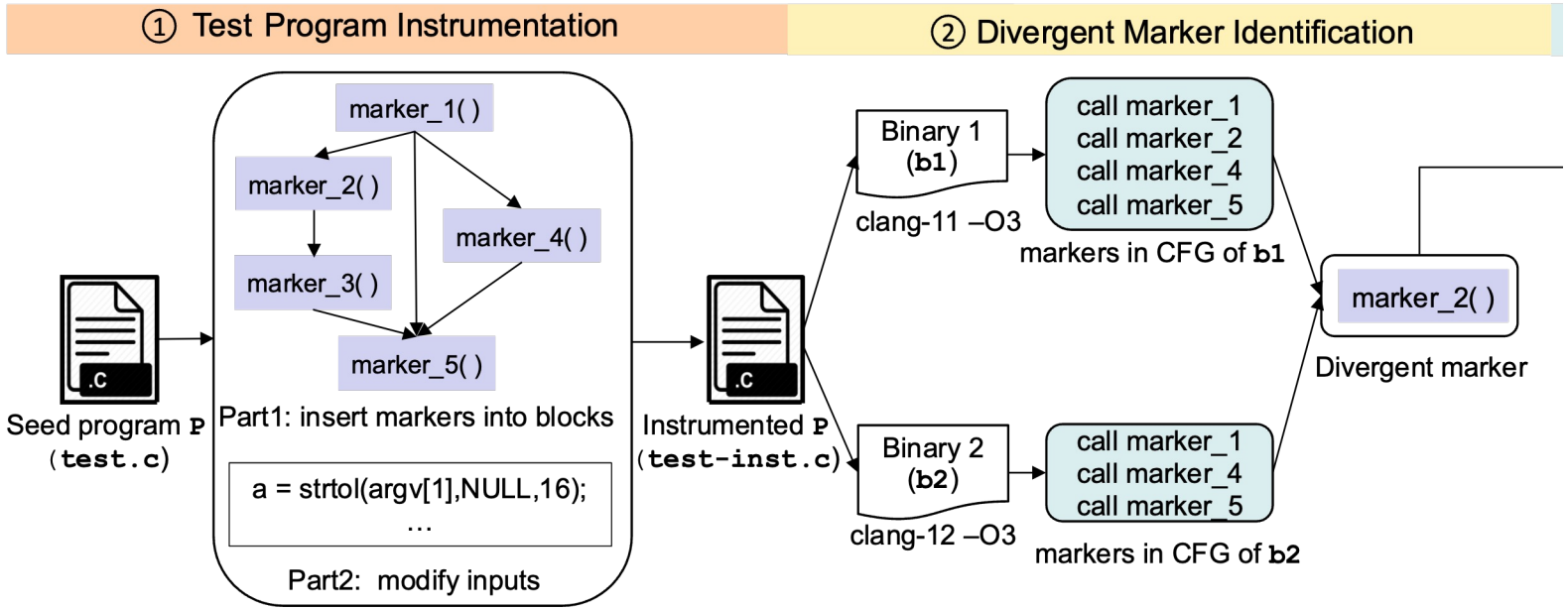



**The marker function:**

```
void __attribute__((noinline)) marker_i() { ++idx; };
```

# Solution: Xdead

- **Key insights: divergence indicates DCE bugs**
  - Identify the divergent portions in binary first
  - leverage symbolic execution to reveal the divergent portion



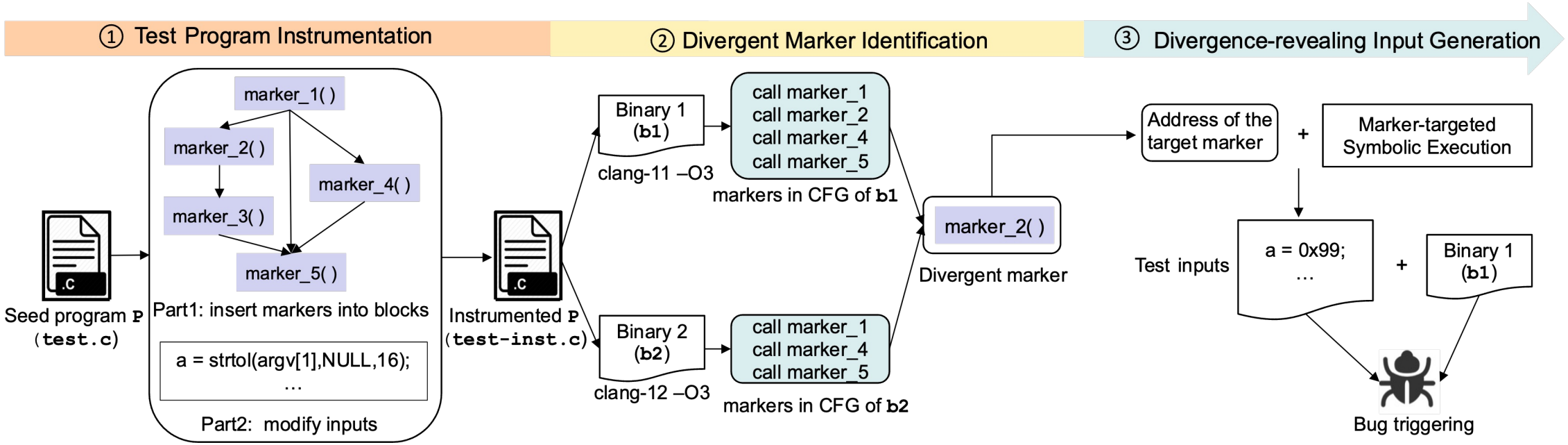

**The marker function:**

```
void __attribute__((noinline)) marker_i() { ++idx; };
```

# Solution: Xdead

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

- **Key insights: divergence indicates DCE bugs**
  - Identify the divergent portions in binary first
  - leverage symbolic execution to reveal the divergent portion




**The marker function:**

```
void __attribute__((noinline)) marker_i() { ++idx; };
```

# Preliminary results

# Preliminary results

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

- **Evaluation setup**
  - Benchmark
    - **10,000** seed program from Csmith
  - Subjects
    - GCC and LLVM
  - Running setting
    - four scenarios under “–O3”

- **Metric**
  - Number of divergent markers
  - Number of bugs

# Preliminary results

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

- **Evaluation setup**
  - Benchmark
    - **10,000** seed program from Csmith
  - Subjects
    - GCC and LLVM
  - Running setting
    - four scenarios under "–O3"

**Table 1: Statistics of divergent markers and test programs**

| Testing Scenarios | Num.Div.b1 | Num.Div.b2 | Num.TP | Per.TP | Ave.M |
|---|---|---|---|---|---|
| GCC-10/11 (-std=c99) | 52,553 | 0 | 5,897 | 58.97% | 8.91 |
| GCC-10/11 (-std=c11) | 49,431 | 0 | 5,758 | 57.58% | 8.59 |
| LLVM-11/12(-std=c99) | 187 | 60 | 70 | 0.007% | 4.12 |
| LLVM-11/12 (-std=c11) | 142 | 57 | 68 | 0.0068% | 2.93 |

- **Metric**
  - Number of divergent markers
  - Number of bugs

# Preliminary results

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

## ➢ Evaluation setup

- Benchmark
  - **10,000** seed program from Csmith
- Subjects
  - GCC and LLVM
- Running setting
  - four scenarios under "–O3"

**Table 1: Statistics of divergent markers and test programs**

| Testing Scenarios | Num.Div.b1 | Num.Div.b2 | Num.TP | Per.TP | Ave.M |
|---|---|---|---|---|---|
| GCC-10/11 (-std=c99) | 52,553 | 0 | 5,897 | 58.97% | 8.91 |
| GCC-10/11 (-std=c11) | 49,431 | 0 | 5,758 | 57.58% | 8.59 |
| LLVM-11/12(-std=c99) | 187 | 60 | 70 | 0.007% | 4.12 |
| LLVM-11/12 (-std=c11) | 142 | 57 | 68 | 0.0068% | 2.93 |

## ➢ Metric

- Number of divergent markers
- Number of bugs

## ➢ Summary

- Found many divergent portions indicating erroneously deleted live code (i.e., wrong compiler optimization opportunities)
- Detected ***Two*** miscompilation bugs in LLVM compilers

# Conclusion

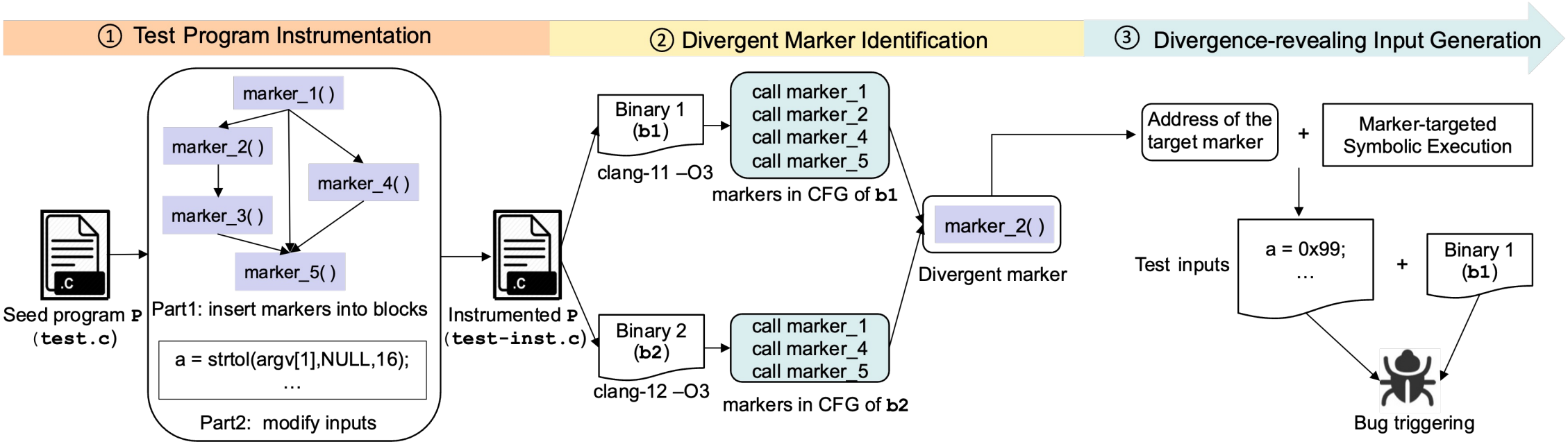
# Conclusion



**Answer: DCE can erroneously delete live code sometimes (Solution: Xdead)**

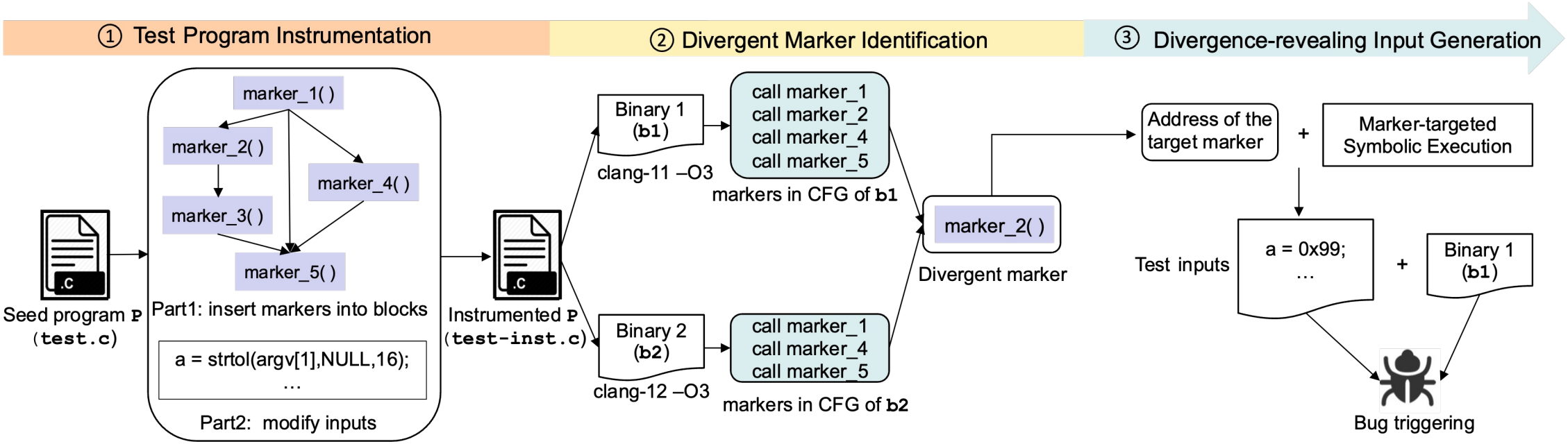
# Conclusion

**Answer: DCE can erroneously delete live code sometimes (Solution: Xdead)**



# Conclusion

**Answer: DCE can erroneously delete live code sometimes (Solution: Xdead)**




- **Future work**
  - Utilize more fine-grained binary analysis to identify fine-grained divergent portions in **Part 1**
  - Improve the efficiency of **Part 3**
    - efficient path exploration

**Paper** **Code**

# Thank you & Questions?

## Beyond a Joke: Dead Code Elimination Can Delete Live Code

**Haoxin Tu**, Lingxiao Jiang, Debin Gao (Singapore Management University)
He Jiang (Dalian University of Technology)

18/04/2024, Lisbon

# Retrospection of motivating example

# Retrospection of motivating example

```
int idx = 0;
int a = 0;
void __attribute__((noinline)) marker_2() { ++idx; }
static void c() { marker_2(); }
void d(int j) { for (;;) ; } // infinite loop
void e() { for(int i = 0; i < 100; m++) d(i);}
void f() {
  e(); // live code here is erroneously deleted
  c();
}
void g() {if (a == 0x99) f();}

int main(int argc, char* argv[]) {
  a = strtol(argv[1], NULL, 16); // when a = 0x99, the bug triggers
  g();
  printf("%d", idx);
  return 0;
} /* affecting -O1 and above on LLVM-11.0.1 downward versions */
```

**Figure 1: A *miscompilation* bug[2] detected by XDEAD**

# Retrospection of motivating example

```
int idx = 0;
int a = 0;
void __attribute__((noinline)) marker_2() { ++idx; }
static void c() { marker_2(); }
void d(int j) { for (;;) ; } // infinite loop
void e() { for(int i = 0; i < 100; m++) d(i);}
void f() {
  e(); // live code here is erroneously deleted
  c();
}
void g() {if (a == 0x99) f();}

int main(int argc, char* argv[]) {
  a = strtol(argv[1], NULL, 16); // when a = 0x99, the bug triggers
  g();
  printf("%d", idx);
  return 0;
} /* affecting -O1 and above on LLVM-11.0.1 downward versions */
```

**Figure 1: A *miscompilation* bug[2] detected by XDEAD**

➢ **Step 1: Test Program Instrumentation**

# Retrospection of motivating example

```
1  int idx = 0;
2  int a = 0;
3  void __attribute__((noinline)) marker_2() { ++idx; }
4  static void c() { marker_2(); }
5  void d(int j) { for (;;) ; } // infinite loop
6  void e() { for(int i = 0; i < 100; m++) d(i);}
7  void f() {
8    e(); // live code here is erroneously deleted
9    c();
10 }
11 void g() {if (a == 0x99) f();}
12
13 int main(int argc, char* argv[]) {
14   a = strtol(argv[1], NULL, 16); // when a = 0x99, the bug triggers
15   g();
16   printf("%d", idx);
17   return 0;
18 } /* affecting -O1 and above on LLVM-11.0.1 downward versions */
```

**Figure 1: A *miscompilation* bug[2] detected by XDEAD**

➢ **Step 1: Test Program Instrumentation**

- Marker injection: Lines 3 and 4

# Retrospection of motivating example

```
1  int idx = 0;
2  int a = 0;
3  void __attribute__((noinline)) marker_2() { ++idx; }
4  static void c() { marker_2(); }
5  void d(int j) { for (;;) ; } // infinite loop
6  void e() { for(int i = 0; i < 100; m++) d(i);}
7  void f() {
8    e(); // live code here is erroneously deleted
9    c();
10 }
11 void g() {if (a == 0x99) f();}
12
13 int main(int argc, char* argv[]) {
14   a = strtol(argv[1], NULL, 16); // when a = 0x99, the bug triggers
15   g();
16   printf("%d", idx);
17   return 0;
18 } /* affecting -O1 and above on LLVM-11.0.1 downward versions */
```

**Figure 1: A *miscompilation* bug[2] detected by XDEAD**

## ➢ Step 1: Test Program Instrumentation

- Marker injection: Lines 3 and 4
- Modify inputs: Lines 1, 14, and 16

# Retrospection of motivating example

```
1  int idx = 0;
2  int a = 0;
3  void __attribute__((noinline)) marker_2() { ++idx; }
4  static void c() { marker_2(); }
5  void d(int j) { for (;;) ; } // infinite loop
6  void e() { for(int i = 0; i < 100; m++) d(i);}
7  void f() {
8    e(); // live code here is erroneously deleted
9    c();
10 }
11 void g() {if (a == 0x99) f();}
12
13 int main(int argc, char* argv[]) {
14   a = strtol(argv[1], NULL, 16); // when a = 0x99, the bug triggers
15   g();
16   printf("%d", idx);
17   return 0;
18 } /* affecting -O1 and above on LLVM-11.0.1 downward versions */
```

**Figure 1: A *miscompilation* bug[2] detected by XDEAD**

### ➢ Step 1: Test Program Instrumentation

- Marker injection: Lines 3 and 4
- Modify inputs: Lines 1, 14, and 16

### ➢ Step 2: Divergent Marker Identification

- "clang-11 –O3 test.c"
- "clang-12 –O3 test.c"
- Get CFG of the binaries and compare the marker existence

# Retrospection of motivating example

```
1  int idx = 0;
2  int a = 0;
3  void __attribute__((noinline)) marker_2() { ++idx; }
4  static void c() { marker_2(); }
5  void d(int j) { for (;;) ; } // infinite loop
6  void e() { for(int i = 0; i < 100; m++) d(i);}
7  void f() {
8    e(); // live code here is erroneously deleted
9    c();
10 }
11 void g() {if (a == 0x99) f();}
12
13 int main(int argc, char* argv[]) {
14   a = strtol(argv[1], NULL, 16); // when a = 0x99, the bug triggers
15   g();
16   printf("%d", idx);
17   return 0;
18 } /* affecting -O1 and above on LLVM-11.0.1 downward versions */
```

**Figure 1: A *miscompilation* bug[2] detected by XDEAD**

### ➢ Step 1: Test Program Instrumentation

- Marker injection: Lines 3 and 4
- Modify inputs: Lines 1, 14, and 16

### ➢ Step 2: Divergent Marker Identification

- "clang-11 –O3 test.c"
- "clang-12 –O3 test.c"
- Get CFG of the binaries and compare the marker existence

### ➢ Step 3: Divergence-revealing Input Generation

- Symbolize the input (i.e., variable a) in the program
- Set the target to find (i.e., the address of function marker_2)
- execute ***Angr*** (a binary symbolic executor)

# Retrospection of motivating example

```
1  int idx = 0;
2  int a = 0;
3  void __attribute__((noinline)) marker_2() { ++idx; }
4  static void c() { marker_2(); }
5  void d(int j) { for (;;) ; } // infinite loop
6  void e() { for(int i = 0; i < 100; m++) d(i);}
7  void f() {
8    e(); // live code here is erroneously deleted
9    c();
10 }
11 void g() {if (a == 0x99) f();}
12
13 int main(int argc, char* argv[]) {
14   a = strtol(argv[1], NULL, 16); // when a = 0x99, the bug triggers
15   g();
16   printf("%d", idx);
17   return 0;
18 } /* affecting -O1 and above on LLVM-11.0.1 downward versions */
```

**Figure 1: A *miscompilation* bug[2] detected by XDEAD**

## ➢ Step 1: Test Program Instrumentation

- Marker injection: Lines 3 and 4
- Modify inputs: Lines 1, 14, and 16

## ➢ Step 2: Divergent Marker Identification

- "clang-11 –O3 test.c"
- "clang-12 –O3 test.c"
- Get CFG of the binaries and compare the marker existence

## ➢ Step 3: Divergence-revealing Input Generation

- Symbolize the input (i.e., variable a) in the program
- Set the target to find (i.e., the address of function marker_2)
- execute ***Angr*** (a binary symbolic executor)

## ➢ Execution results

- "Found a solution!": e.g., 0x99
- "No solution found": not a DCE bug or Angr encounter difficulties

# Related work

[1] Tu, Haoxin, He Jiang, Xiaochen Li, Zhilei Ren, Zhide Zhou, and Lingxiao Jiang. "RemGen: Remanufacturing a random program generator for compiler testing." In *IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*, pp. 529-540, 2022.

# Related work

- **Compiler testing studies [1]**

[1] Tu, Haoxin, He Jiang, Xiaochen Li, Zhilei Ren, Zhide Zhou, and Lingxiao Jiang. "RemGen: Remanufacturing a random program generator for compiler testing." In *IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*, pp. 529-540, 2022.

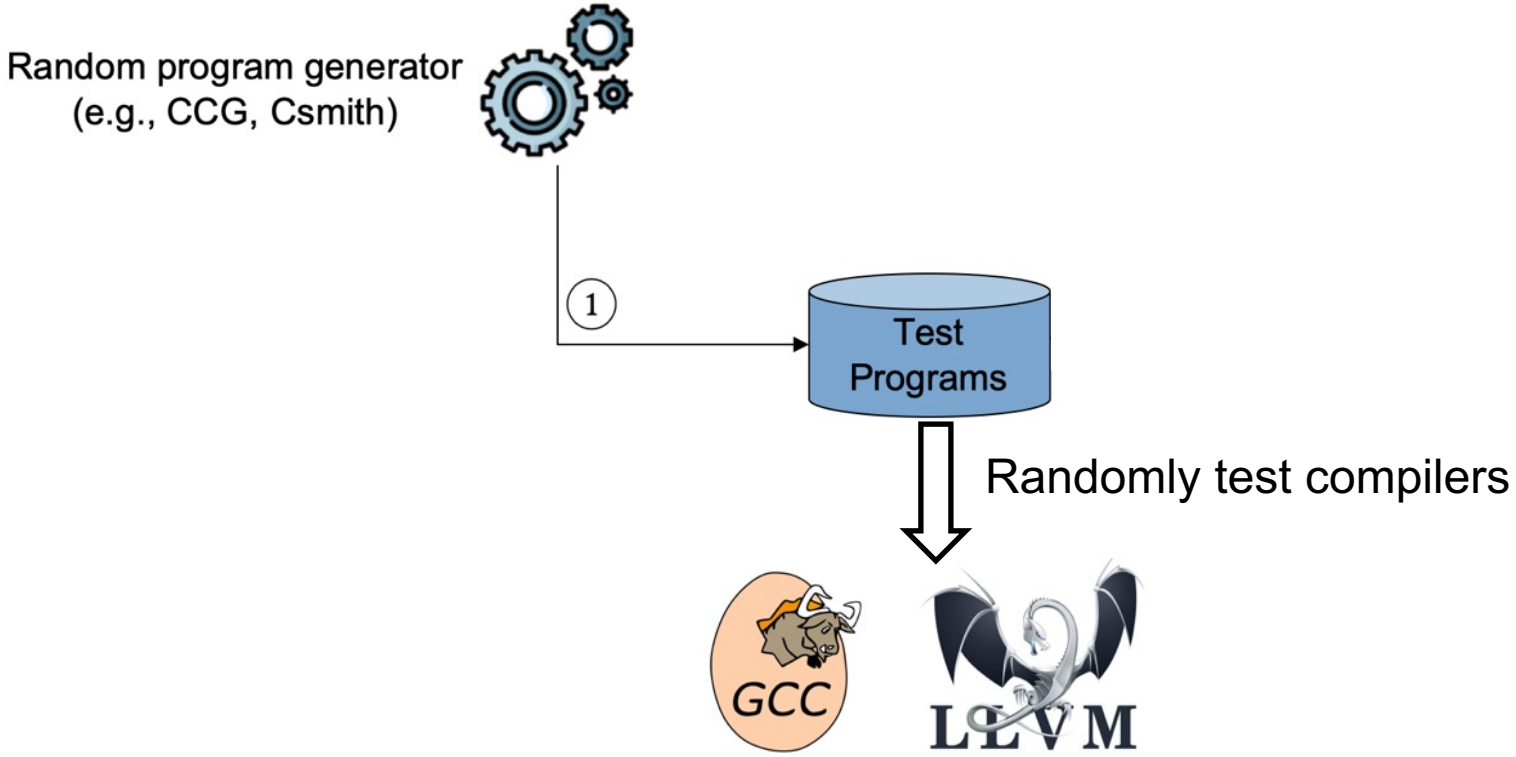
# Related work

- **Compiler testing studies [1]**
  - Generation-based: Csmith, YARPGen, etc.
  - Mutation-based: Orion, Athena, and Hermes, etc.

[1] Tu, Haoxin, He Jiang, Xiaochen Li, Zhilei Ren, Zhide Zhou, and Lingxiao Jiang. "RemGen: Remanufacturing a random program generator for compiler testing." In *IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*, pp. 529-540, 2022.

# Related work

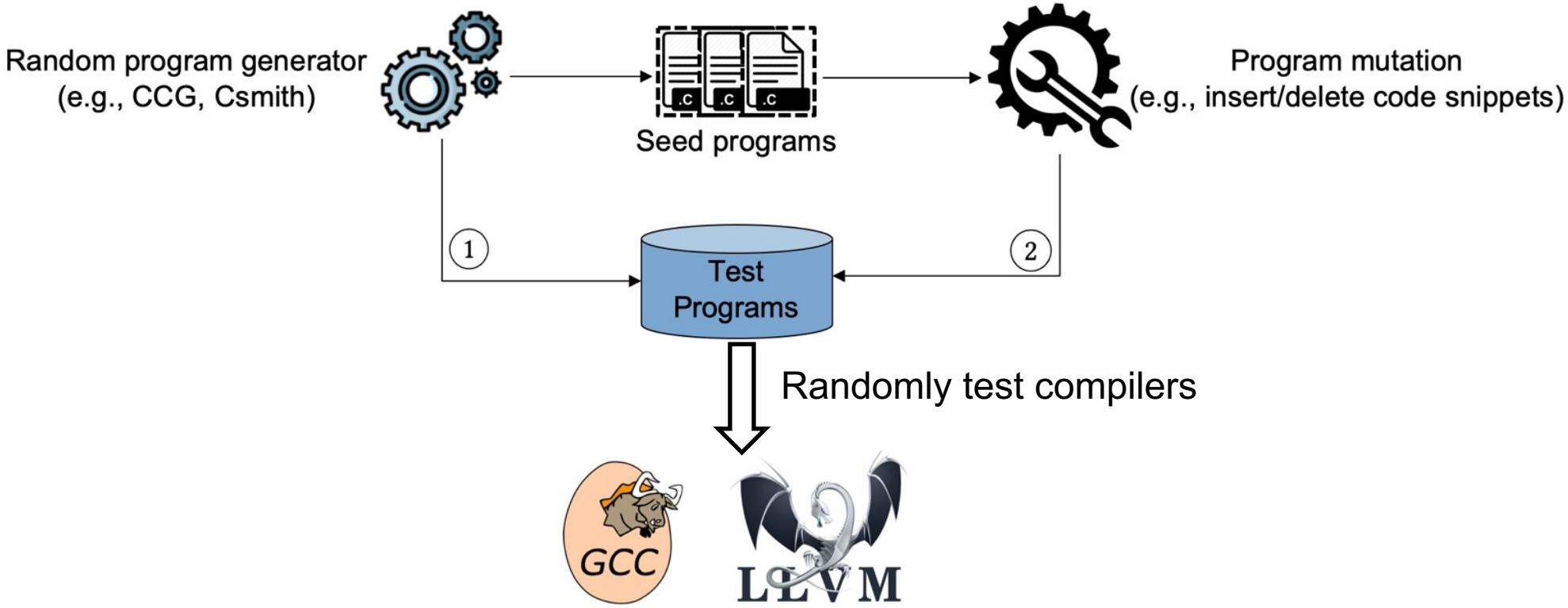
- **Compiler testing studies [1]**
  - Generation-based: Csmith, YARPGen, etc.
  - Mutation-based: Orion, Athena, and Hermes, etc.




[1] Tu, Haoxin, He Jiang, Xiaochen Li, Zhilei Ren, Zhide Zhou, and Lingxiao Jiang. "RemGen: Remanufacturing a random program generator for compiler testing." In *IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*, pp. 529-540, 2022.

# Related work

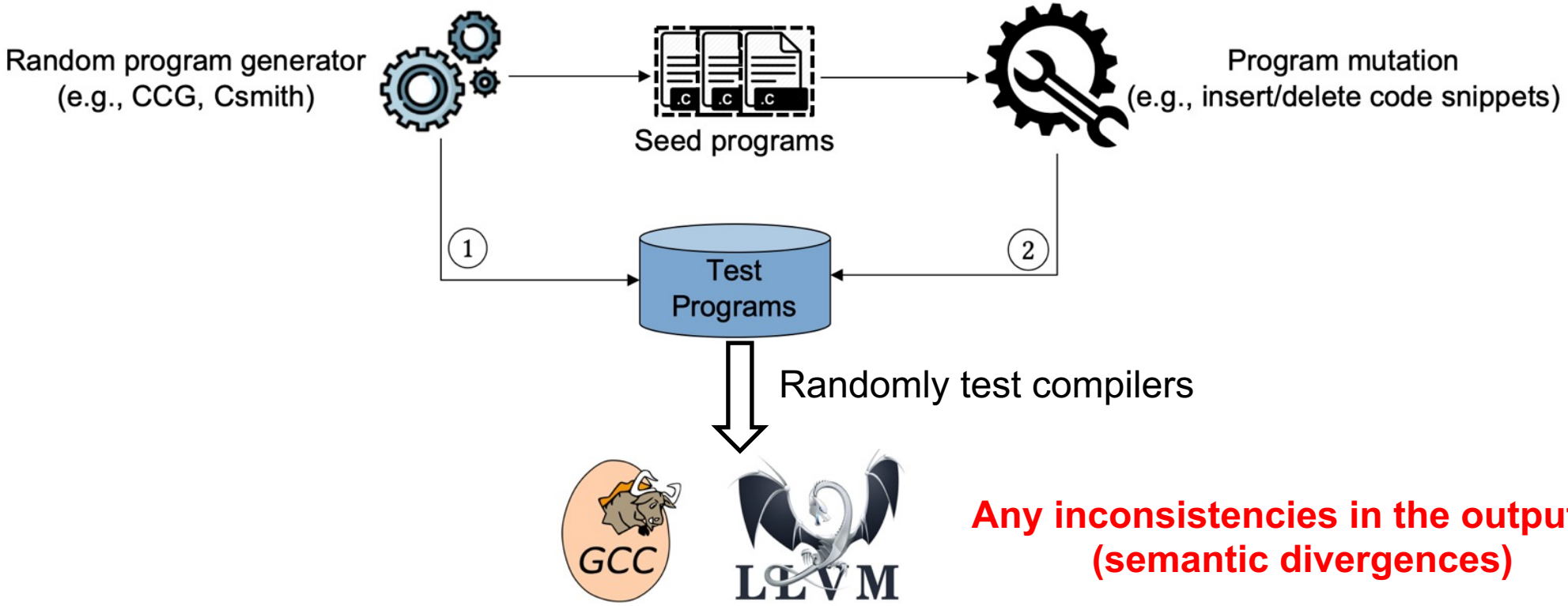
- **Compiler testing studies [1]**
  - Generation-based: Csmith, YARPGen, etc.
  - Mutation-based: Orion, Athena, and Hermes, etc.




[1] Tu, Haoxin, He Jiang, Xiaochen Li, Zhilei Ren, Zhide Zhou, and Lingxiao Jiang. "RemGen: Remanufacturing a random program generator for compiler testing." In *IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*, pp. 529-540, 2022.

# Related work

- **Compiler testing studies [1]**
  - Generation-based: Csmith, YARPGen, etc.
  - Mutation-based: Orion, Athena, and Hermes, etc.




**Any inconsistencies in the outputs?**
**(semantic divergences)**

[1] Tu, Haoxin, He Jiang, Xiaochen Li, Zhilei Ren, Zhide Zhou, and Lingxiao Jiang. "RemGen: Remanufacturing a random program generator for compiler testing." In *IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*, pp. 529-540, 2022.

# Future work (1/2)

# Future work (1/2)



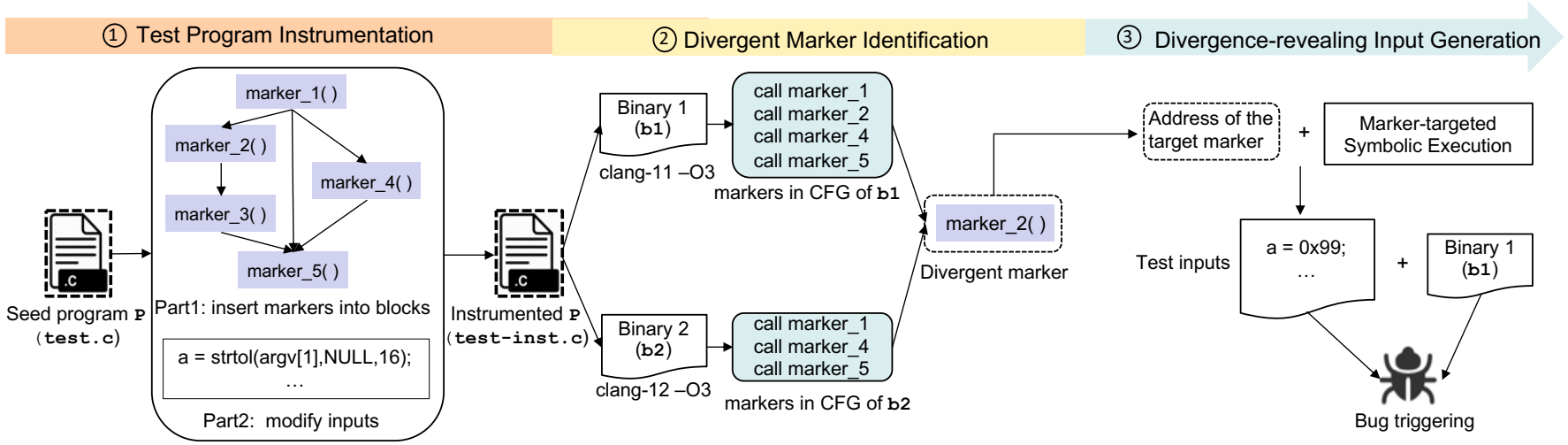
- **Insights**
  - ✓ Reducing the search space of symbolic exploration could make targeted symbolic execution more efficient

# Future work (1/2)

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

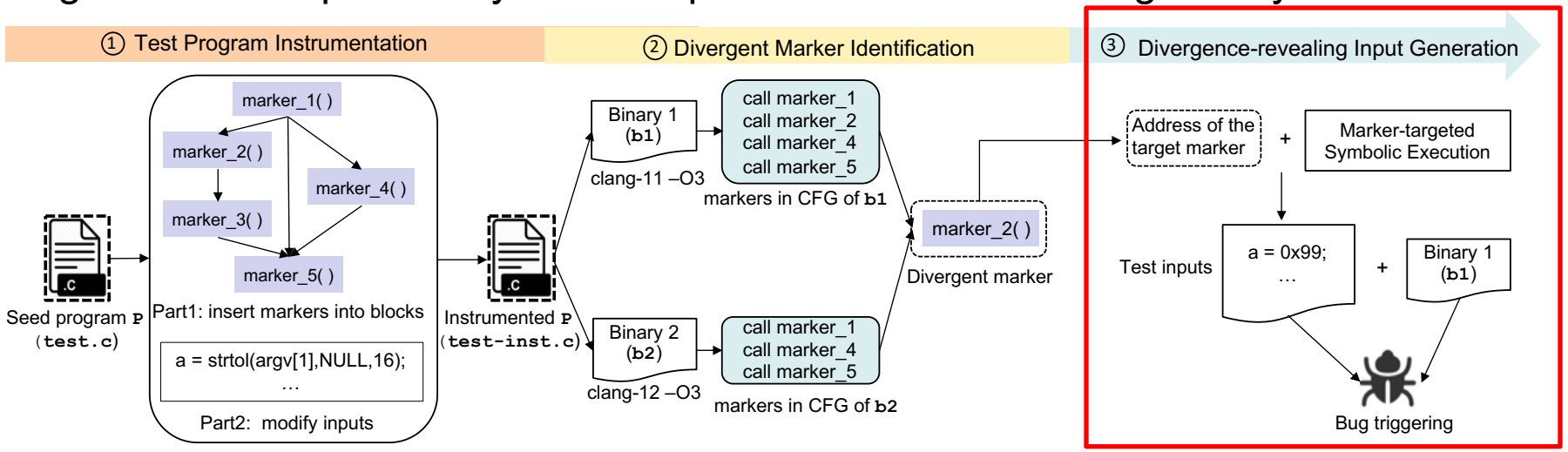
## ➢ Insights

✓ Reducing the search space of symbolic exploration could make targeted symbolic execution more efficient



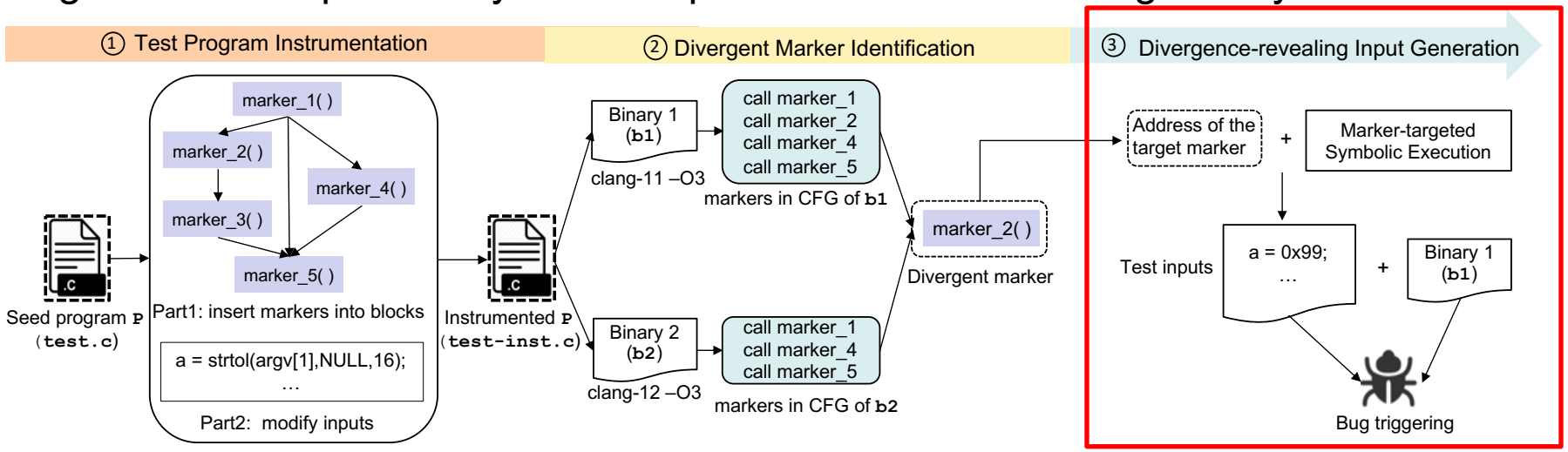
# Future work (1/2)

- **Insights**
  - ✓ Reducing the search space of symbolic exploration could make targeted symbolic execution more efficient



# Future work (1/2)

- **Insights**
  - ✓ Reducing the search space of symbolic exploration could make targeted symbolic execution more efficient

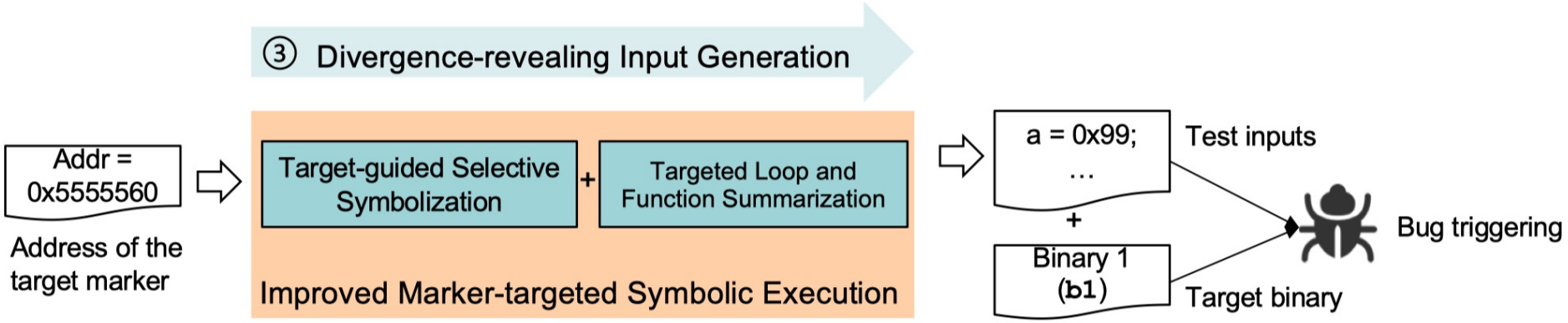



- **Challenge 1: how to select important variables to be symbolized?**
  - Not every variable contributes equally to revealing the divergent portion in binaries
- **Challenge 2: how to handle loops or function calls involved with symbolic variables?**
  - Such loops or functions make the target symbolic execution inefficient

# Future work (2/2)

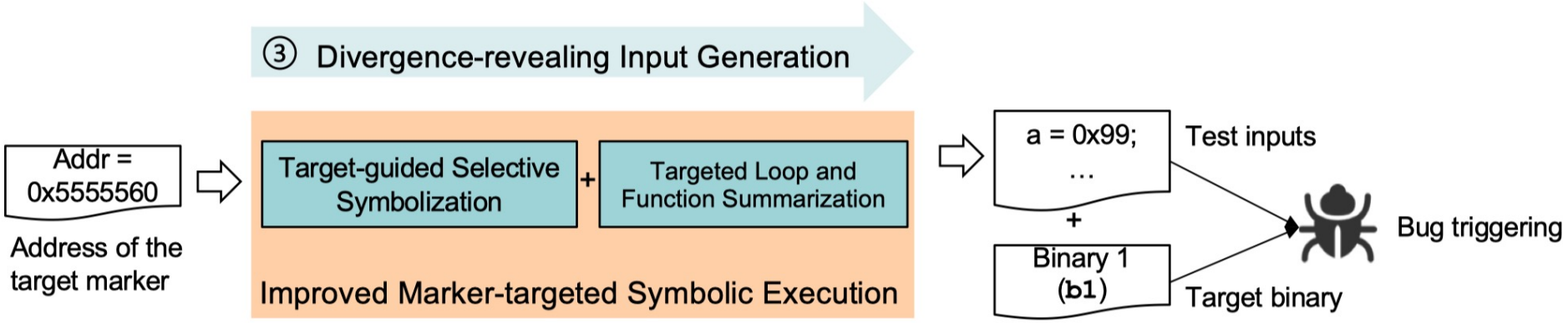
# Future work (2/2)

- **Planned solution: efficient targeted symbolic execution**



# Future work (2/2)

- **Planned solution: efficient targeted symbolic execution**




- **Addressing challenge 1: Target-guided selective symbolization**
  - Data flow analysis to select variables that are directly to go through the divergent functions
- **Addressing challenge 2: Targeted loop and function summarizations**
  - Handle loops and functions that involve symbolic variables