Improving Directed Fuzzing by projecting string domain input to finite state machine

Team 2 Final Presentation

Tae Eun Kim, Jaeho Kim, Kihwan Kim, Seunghyeon Jeong

Background

Greybox Fuzzing - Explores the program guided by the code coverage.

Directed Fuzzing - Aims to produce input that reaches the given target.

Recap of proposal

Recap of proposal

Problem: Greybox fuzzers struggle with the absence of code coverage.

Limitation of Greybox Fuzzing

- Conditions with no intermediate code coverage is challenging

- Ex) String involved conditions (e.g. strcmp, strstr, ...)

Limitation of Greybox Fuzzing

- Conditions with no intermediate code coverage is challenging
- Ex) String involved conditions (e.g. strcmp, strstr, ...)

```
 1: int main(int argc, char* argv[]) {
 2:     if (argc < 2)
 3:         return 1;
 4:
 5:     if (strcmp(argv[1], "HELLO")) {
 6:         Crash();
 7:         printf("Hello World!");
 8:         return 0;
 9:     }
10:
11:     OK();
12:     return 0;
13: }
```

Limitation of Greybox Fuzzing

- Conditions with no intermediate code coverage is challenging
- Ex) String involved conditions (e.g. strcmp, strstr, ...)

```
 1: int main(int argc, char* argv[]) {
 2:     if (argc < 2)
 3:         return 1;
 4:
 5:     if (strcmp(argv[1], "HELLO")) {
 6:         Crash();
 7:         printf("Hello World!");
 8:         return 0;
 9:     }
10:
11:     OK();
12:     return 0;
13: }
```

2^{40}

Recap of proposal

Problem: Greybox fuzzers struggle with the absence of code coverage.

Recap of proposal

Problem: Greybox fuzzers struggle with the absence of code coverage.

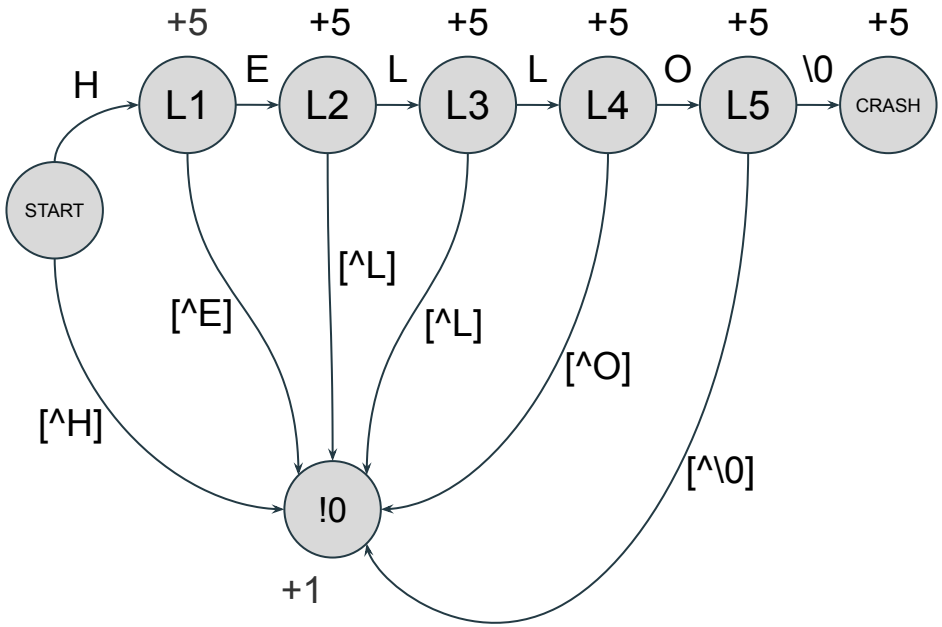
Solution: Use Finite State Machine to give intermediate code coverage.

Example 1

```
int main(int argc, char* argv[]) {
    if (argc < 2)
        return 1;

    if (strcmp(argv[1], "HELLO")) {
        Crash();
        printf("Hello World!");
        return 0;
    }

    OK();
    return 0;
}
```

Example 1

```
int main(int argc, char* argv[]) {
    if (argc < 2)
        return 1;

    if (strcmp(argv[1], "HELLO")) {
        Crash();
        printf("Hello World!");
        return 0;
    }

    OK();
    return 0;
}
```



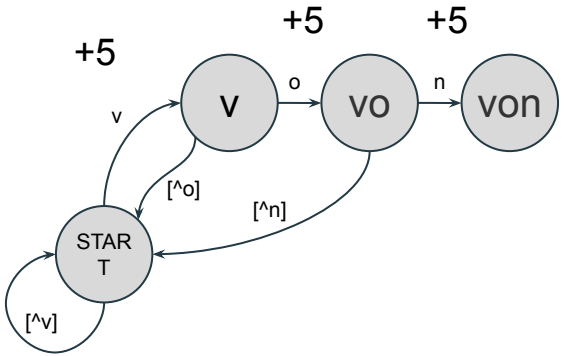
Input: "BYE", "HE", "HELL", …

Example 2

```
int main(int argc, char* argv[]) {
    char *abbr;
    if (argc < 2)
        return 1;

    if (abbr = strstr(argv[1], "von")) {
        Crash();
        return 0;
    }

    OK();
    return 0;
}
```

Example 2

```
int main(int argc, char* argv[]) {
    char *abbr;
    if (argc < 2)
        return 1;

    if (abbr = strstr(argv[1], "von")) {
        Crash();
        return 0;
    }

    OK();
    return 0;
}
```

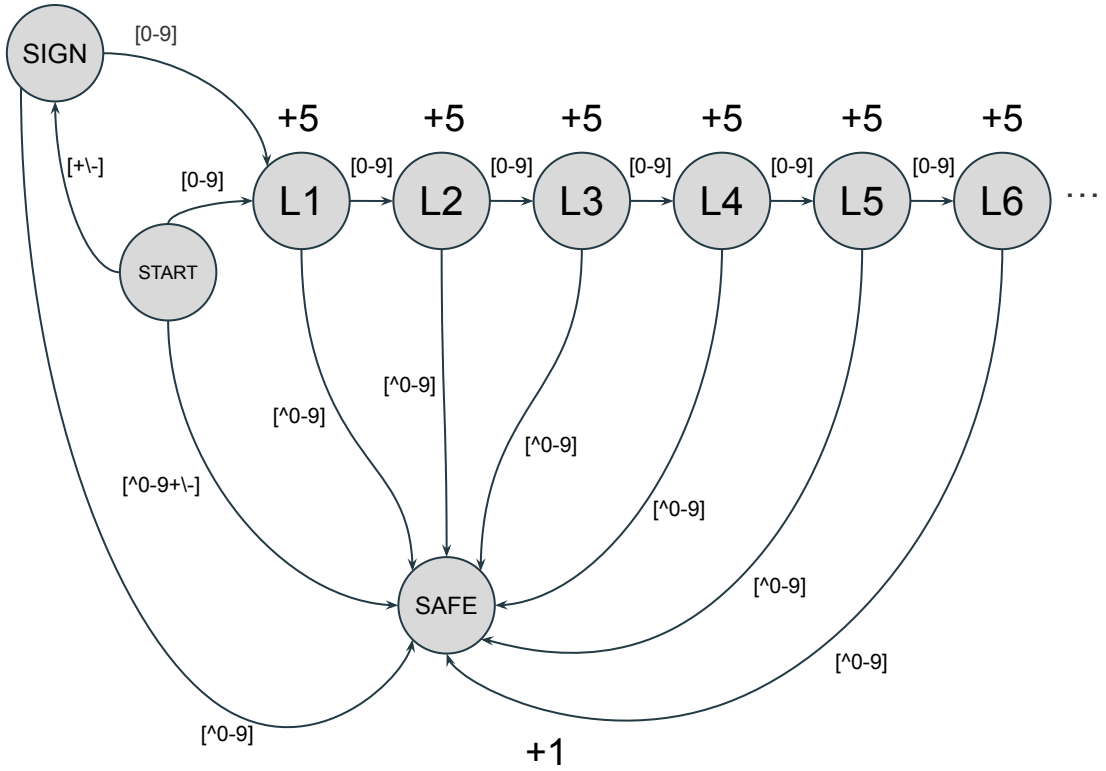


Example 3

```
int main(int argc, char* argv[]) {
    if (argc < 2)
        return 1;

    if (atoi(argv[1]) > 90000) {
        Crash();
        return 0;
    }

    OK();
    return 0;
}
```

Example 3

```
int main(int argc, char* argv[]) {
    if (argc < 2)
        return 1;

    if (atoi(argv[1]) > 90000) {
        Crash();
        return 0;
    }

    OK();
    return 0;
}
```



Recap of proposal

Problem: Greybox fuzzers struggle with the absence of code coverage.

Solution: Use Finite State Machine to give intermediate code coverage.

Scope: Program with string input domain.

Baseline: AFLGo

Implementation

How to instrument & How to get coverage feedback

AFL

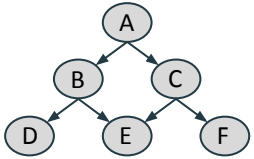
Utilize a bitmap mapped with all branch edges

Unseen coverage of branch edge -> Interesting!!

AFL

Utilize a bitmap mapped with all branch edges

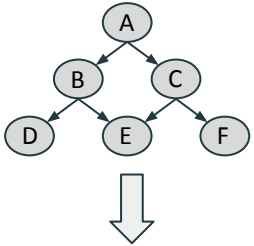
Unseen coverage of branch edge -> Interesting!!



AFL

Utilize a bitmap mapped with all branch edges

Unseen coverage of branch edge -> Interesting!!

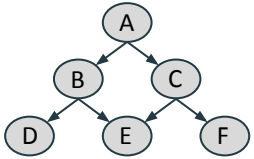


AB	AC	BD
BE	CE	CF

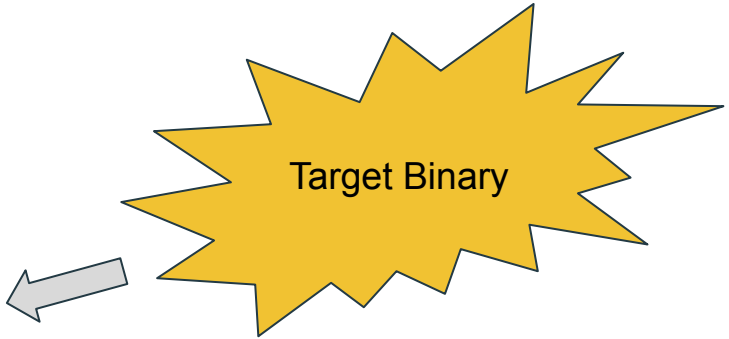
AFL

Utilize a bitmap mapped with all branch edges

Unseen coverage of branch edge -> Interesting!!



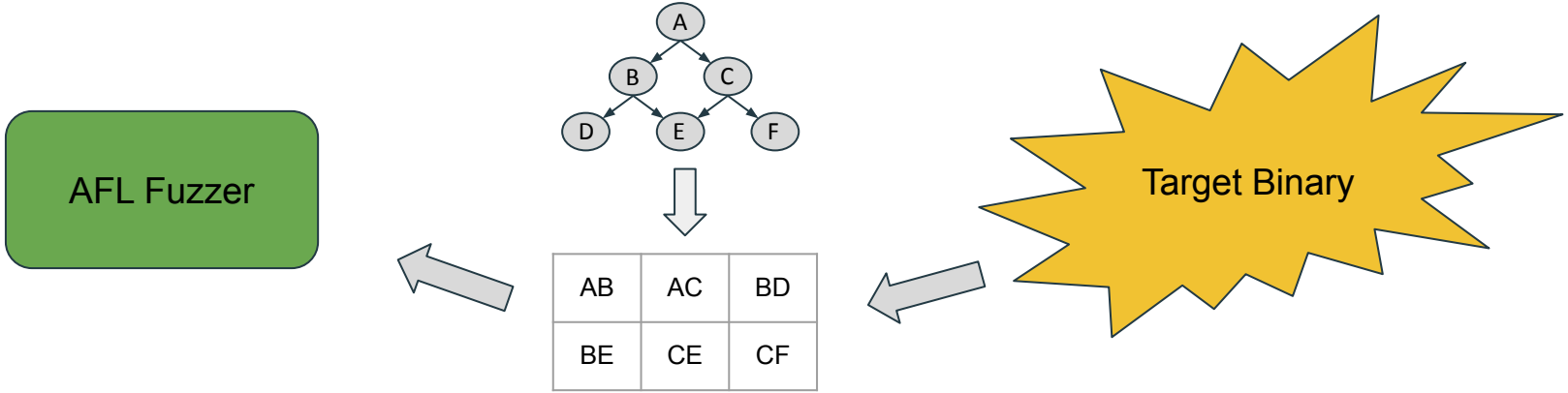
AB	AC	BD
BE	CE	CF



AFL

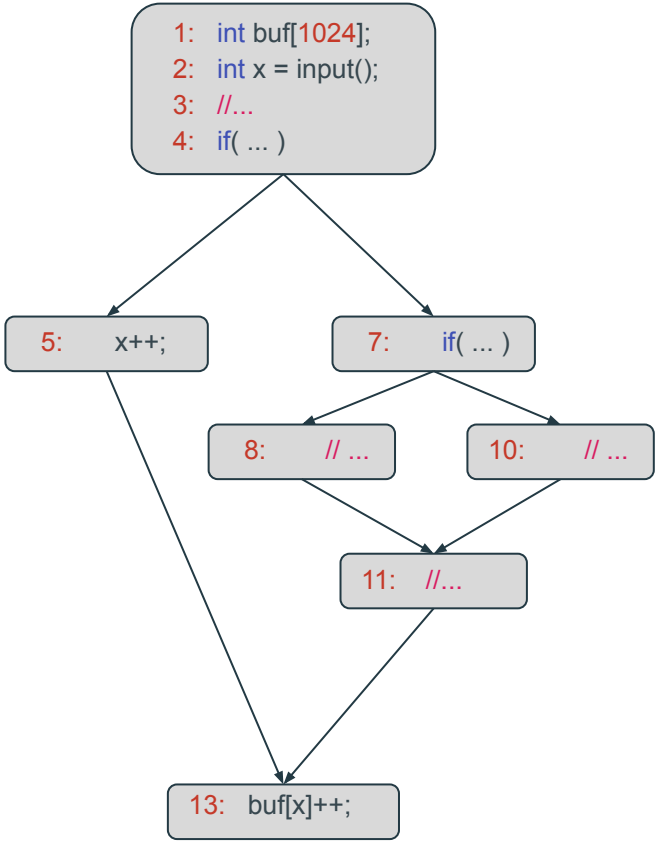
Utilize a bitmap mapped with all branch edges

Unseen coverage of branch edge -> Interesting!!

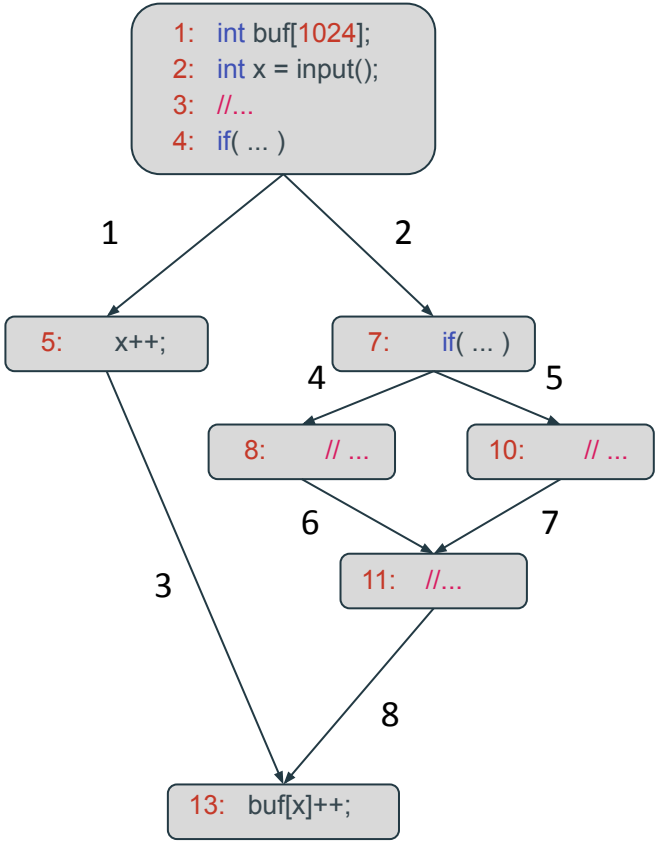


```
 1:  int buf[1024];
 2:  int x = input();
 3:  //...
 4:  if( ... )
 5:      x++;
 6:  else {
 7:      if( ... )
 8:       // ...
 9:      else
10:       // ...
11:   //...
12:  }
13:  buf[x]++;
```

```
1:  int buf[1024];
2:  int x = input();
3:  //...
4:  if( ... )
5:      x++;
6:  else {
7:      if( ... )
8:       // ...
9:      else
10:      // ...
11:  //...
12:  }
13:  buf[x]++;
```

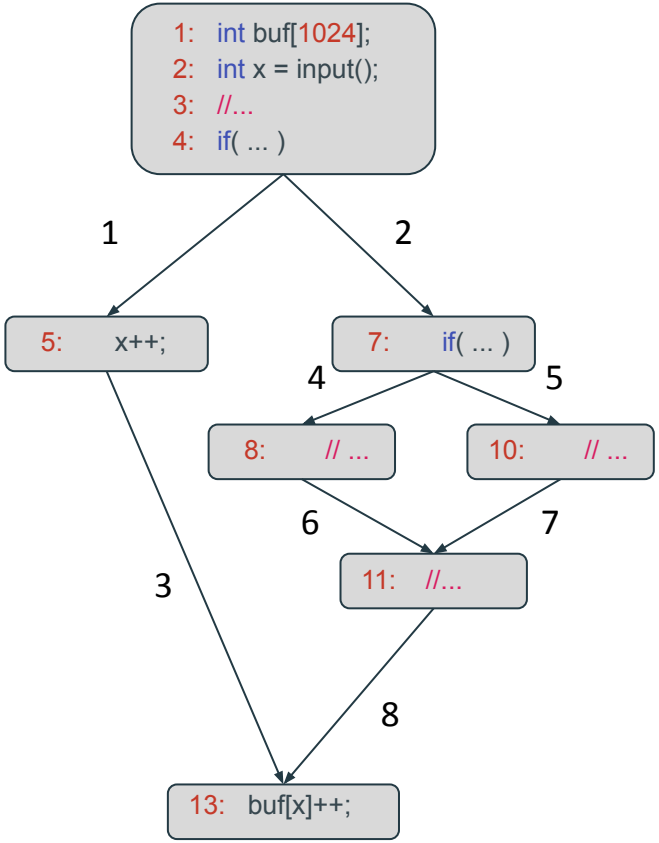


```
1:  int buf[1024];
2:  int x = input();
3:  //...
4:  if( ... )
5:      x++;
6:  else {
7:      if( ... )
8:       // ...
9:      else
10:      // ...
11:  //...
12:  }
13:  buf[x]++;
```



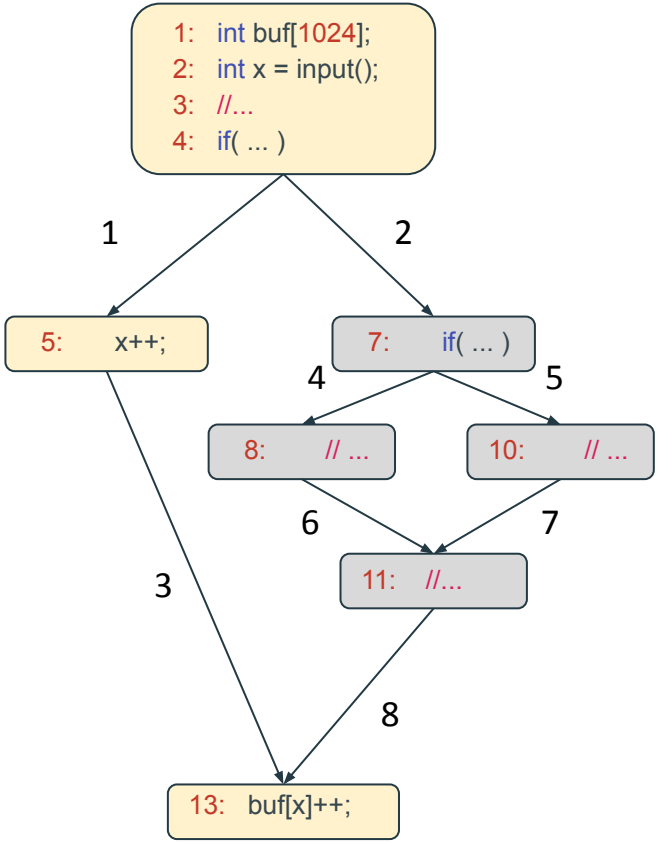
```
1:  int buf[1024];
2:  int x = input();
3:  //...
4:  if( ... )
5:      x++;
6:  else {
7:      if( ... )
8:       // ...
9:      else
10:      // ...
11:  //...
12:  }
13:  buf[x]++;
```

Input A



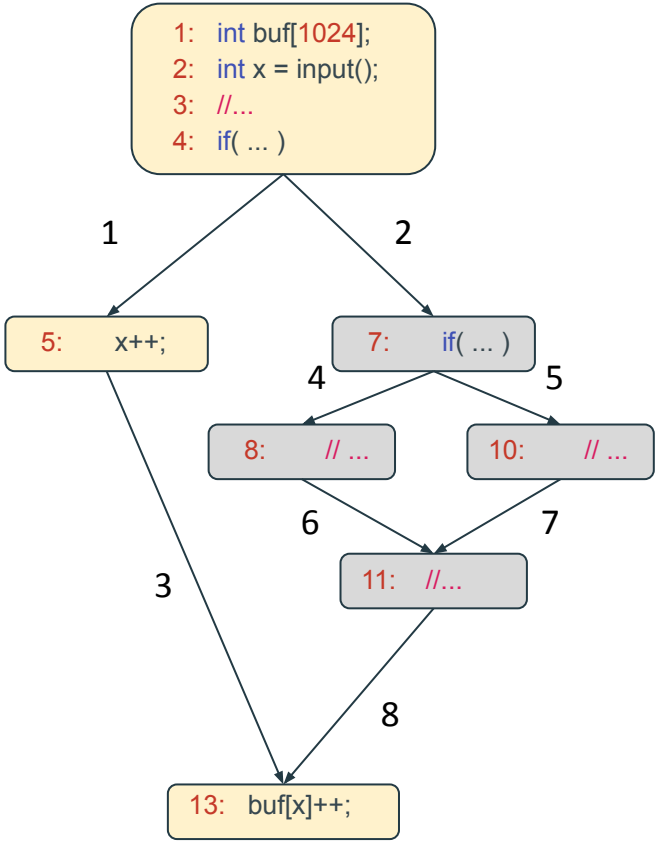
```
1:  int buf[1024];
2:  int x = input();
3:  //...
4:  if( ... )
5:      x++;
6:  else {
7:      if( ... )
8:       // ...
9:      else
10:      // ...
11:  //...
12:  }
13:  buf[x]++;
```

Input A



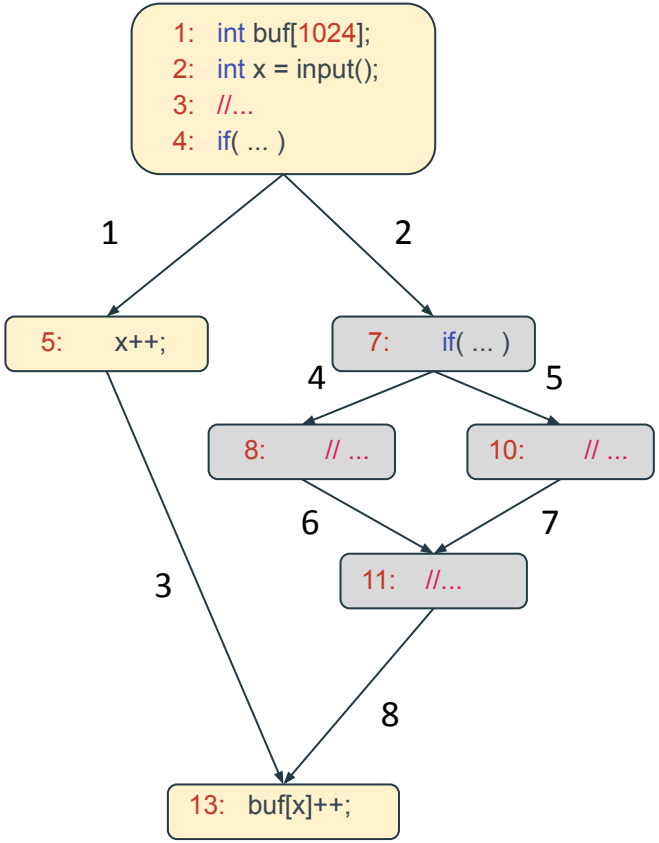
```
1:  int buf[1024];
2:  int x = input();
3:  //...
4:  if( ... )
5:      x++;
6:  else {
7:      if( ... )
8:       // ...
9:      else
10:      // ...
11:  //...
12:  }
13:  buf[x]++;
```

Input A {1,3}



```
1:  int buf[1024];
2:  int x = input();
3:  //...
4:  if( ... )
5:      x++;
6:  else {
7:      if( ... )
8:       // ...
9:      else
10:      // ...
11:  //...
12:  }
13:  buf[x]++;
```

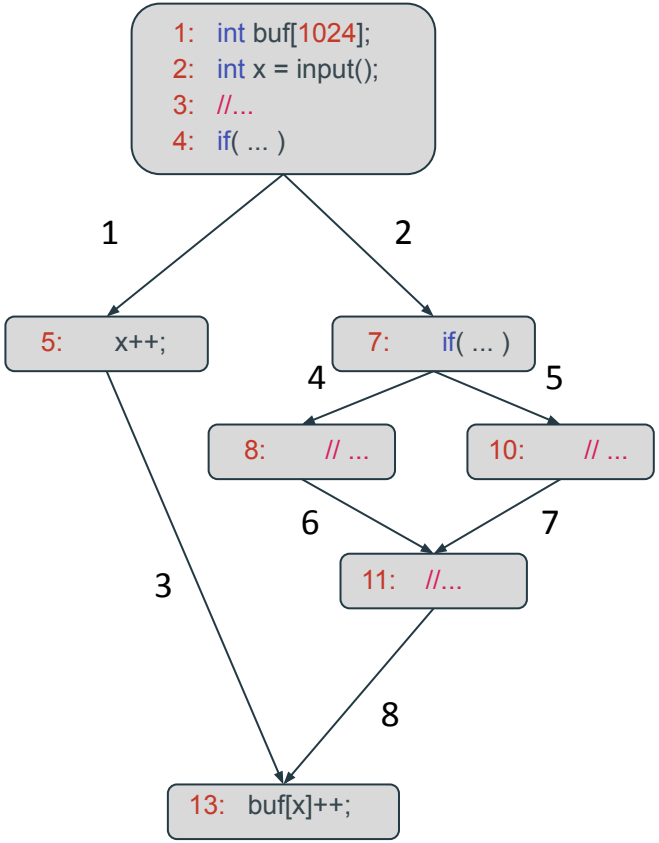
Seed A {1,3}



```
1:  int buf[1024];
2:  int x = input();
3:  //...
4:  if( ... )
5:      x++;
6:  else {
7:      if( ... )
8:       // ...
9:      else
10:      // ...
11:  //...
12:  }
13:  buf[x]++;
```

Seed A {1,3}

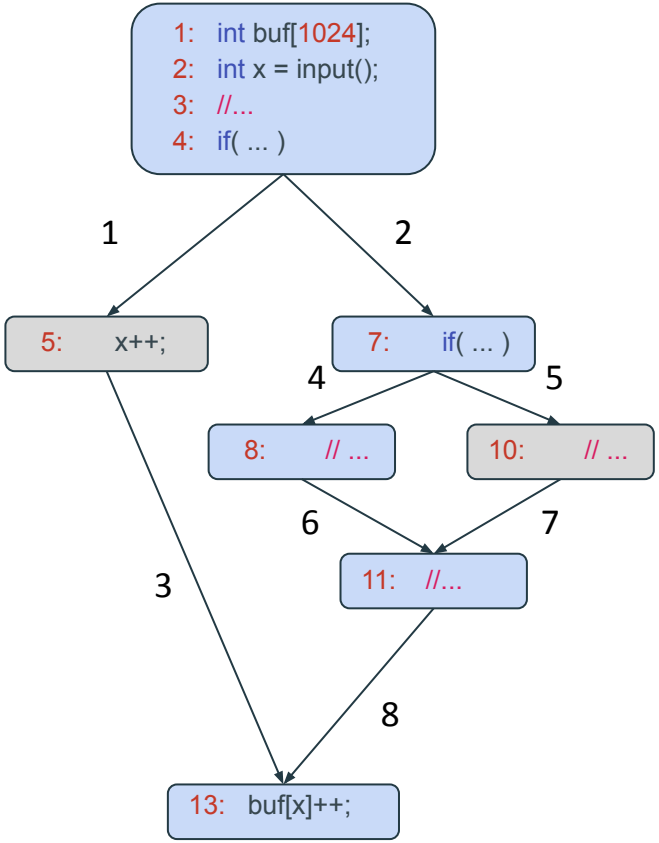
Input B



```
1:  int buf[1024];
2:  int x = input();
3:  //...
4:  if( ... )
5:      x++;
6:  else {
7:      if( ... )
8:       // ...
9:      else
10:      // ...
11:  //...
12:  }
13:  buf[x]++;
```

Seed A {1,3}

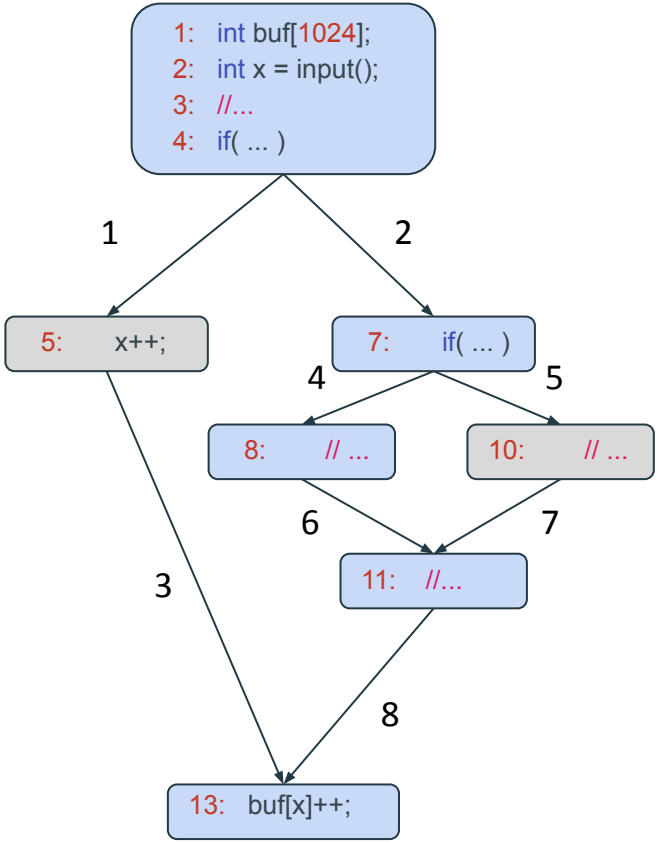
Input B



```
1:  int buf[1024];
2:  int x = input();
3:  //...
4:  if( ... )
5:      x++;
6:  else {
7:      if( ... )
8:       // ...
9:      else
10:      // ...
11:  //...
12:  }
13:  buf[x]++;
```

Seed A {1,3}

Input B {2,4,6,8}

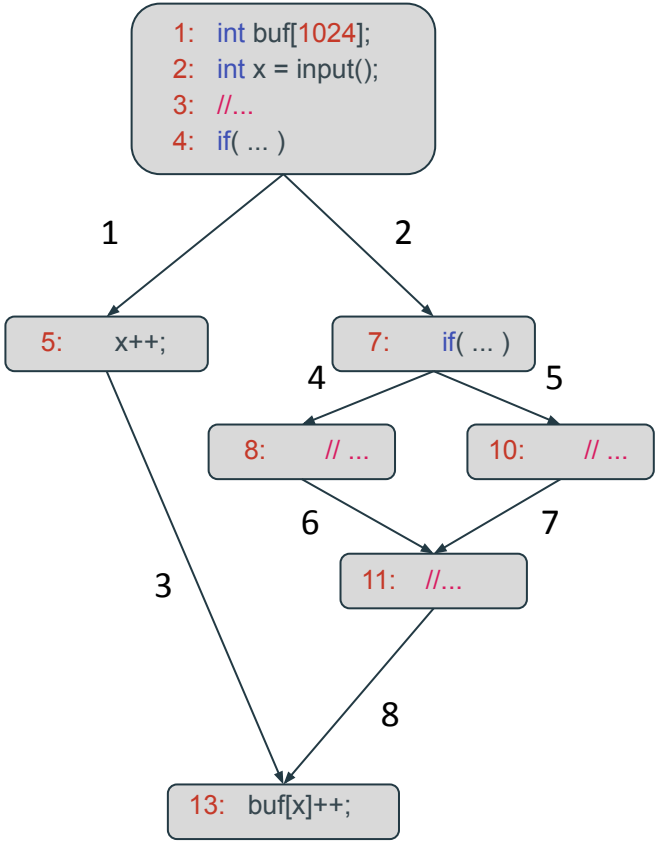


```
1:  int buf[1024];
2:  int x = input();
3:  //...
4:  if( ... )
5:      x++;
6:  else {
7:      if( ... )
8:       // ...
9:      else
10:      // ...
11:  //...
12:  }
13:  buf[x]++;
```

Seed A {1,3}

Seed B {2,4,6,8}

Input C

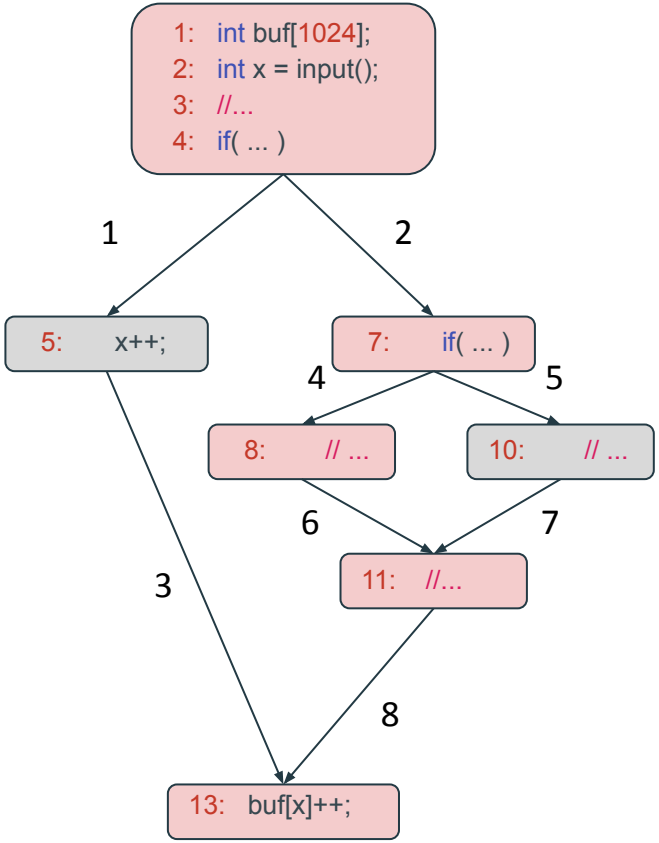


```
1:  int buf[1024];
2:  int x = input();
3:  //...
4:  if( ... )
5:      x++;
6:  else {
7:      if( ... )
8:       // ...
9:      else
10:      // ...
11:  //...
12:  }
13:  buf[x]++;
```

Seed A {1,3}

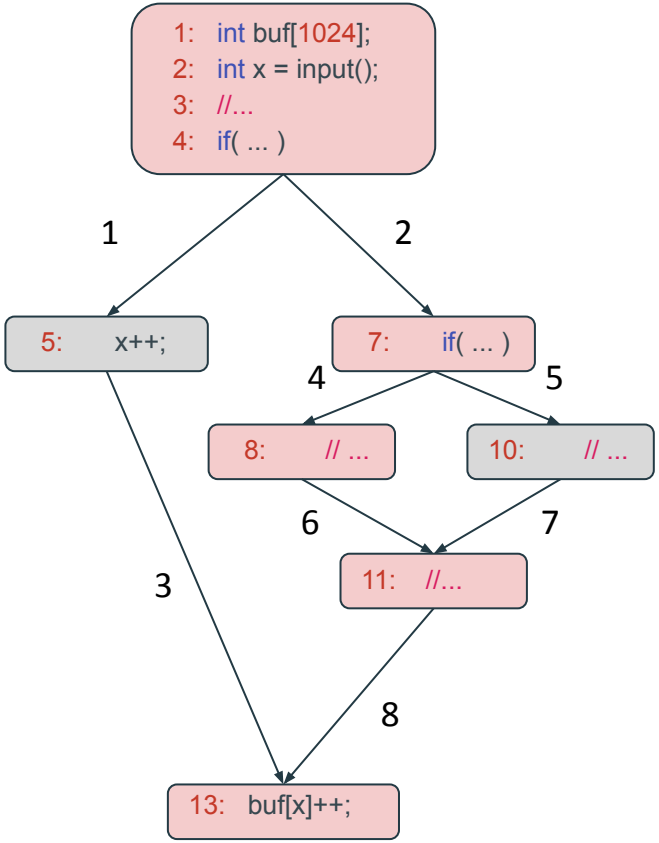
Seed B {2,4,6,8}

Input C



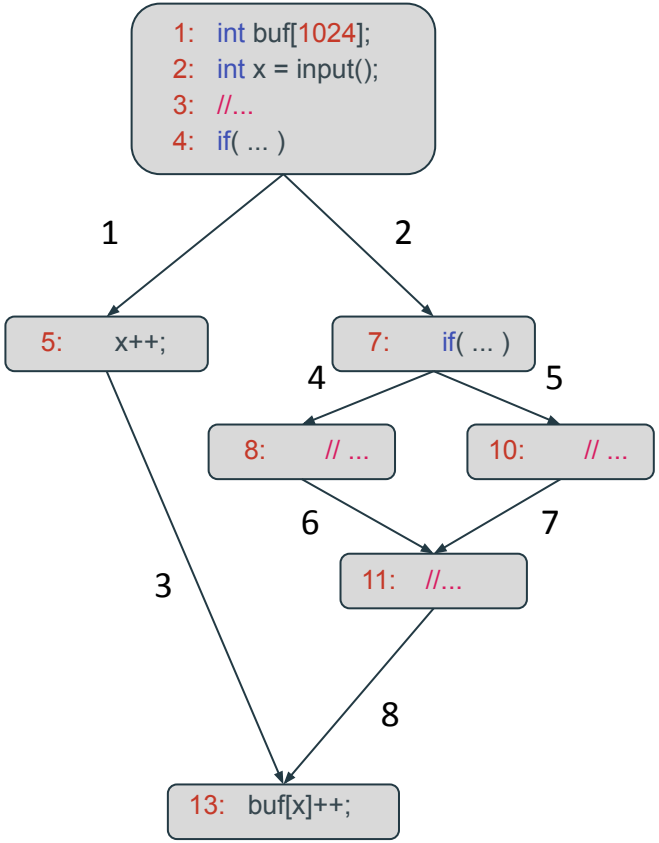
```
1:  int buf[1024];
2:  int x = input();
3:  //...
4:  if( ... )
5:      x++;
6:  else {
7:      if( ... )
8:       // ...
9:      else
10:      // ...
11:  //...
12:  }
13:  buf[x]++;
```

Seed A	{1,3}
Seed B	{2,4,6,8}
Input C	{2,4,6,8}

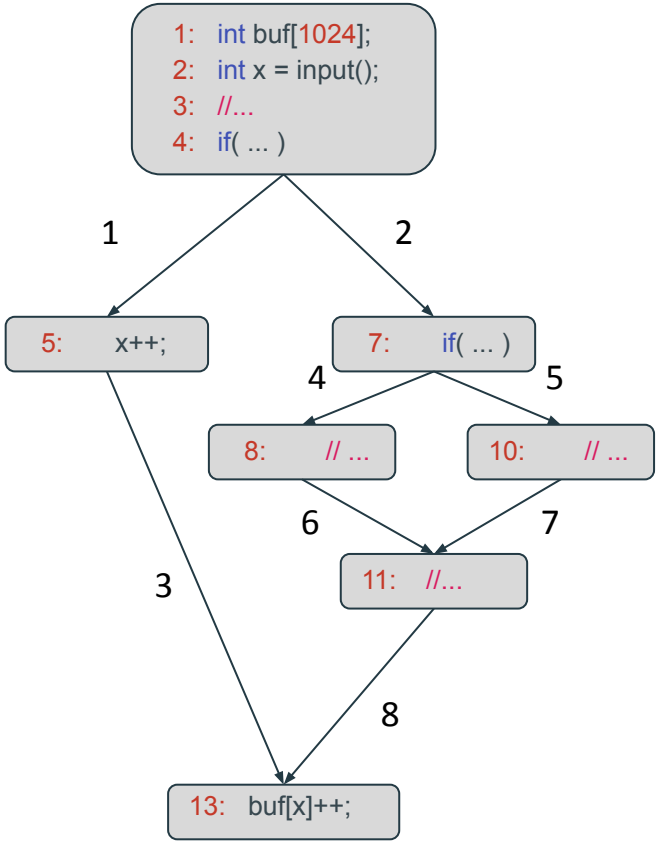


```
1:  int buf[1024];
2:  int x = input();
3:  //...
4:  if( ... )
5:      x++;
6:  else {
7:      if( ... )
8:       // ...
9:      else
10:      // ...
11:  //...
12:  }
13:  buf[x]++;
```

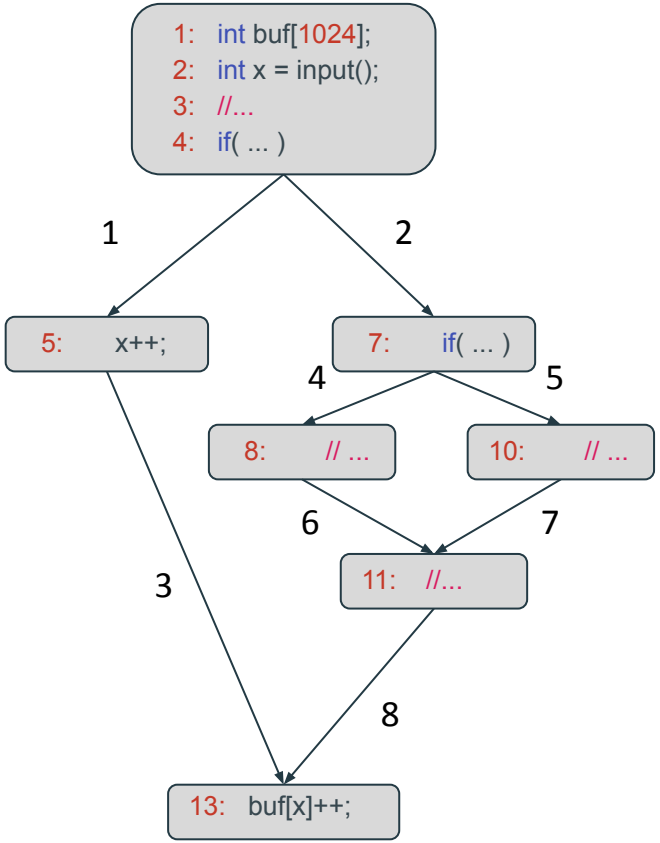
Seed A	{1,3}
Seed B	{2,4,6,8}
Input C	{2,4,6,8}



AFL & AFLGo

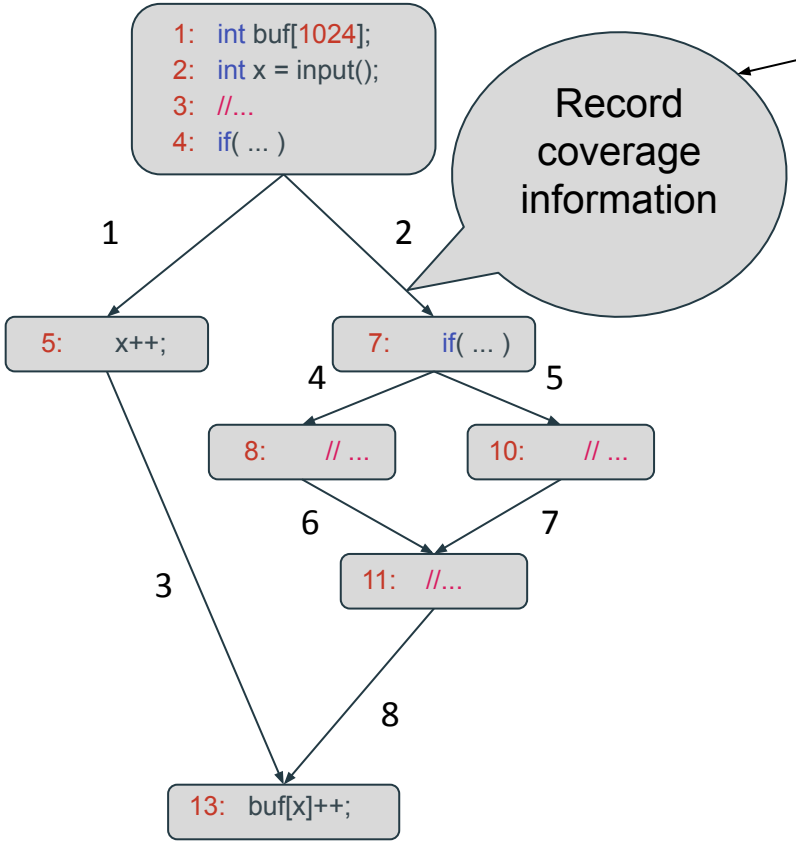


AFL & AFLGo



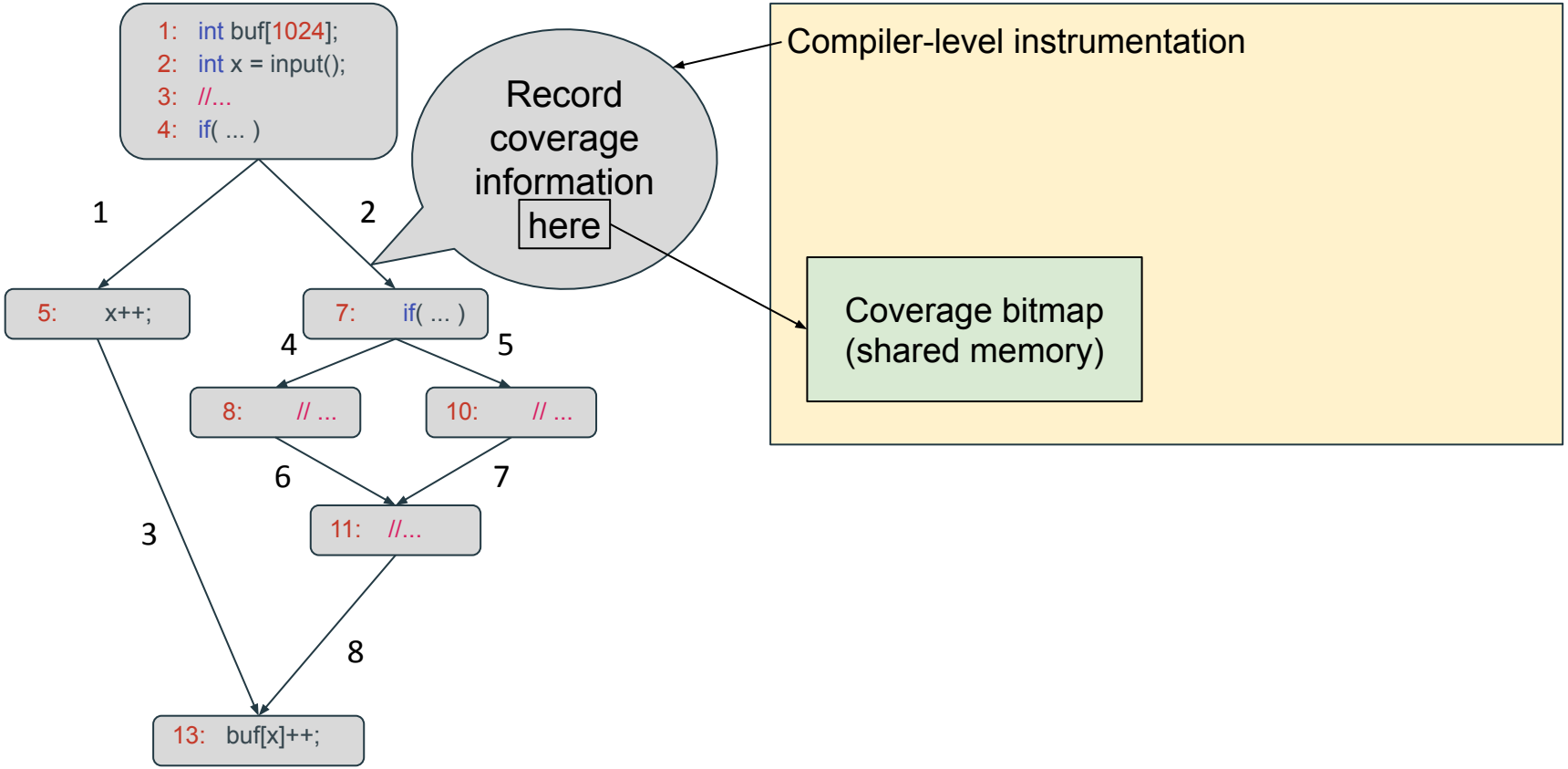
Compiler-level instrumentation

AFL & AFLGo

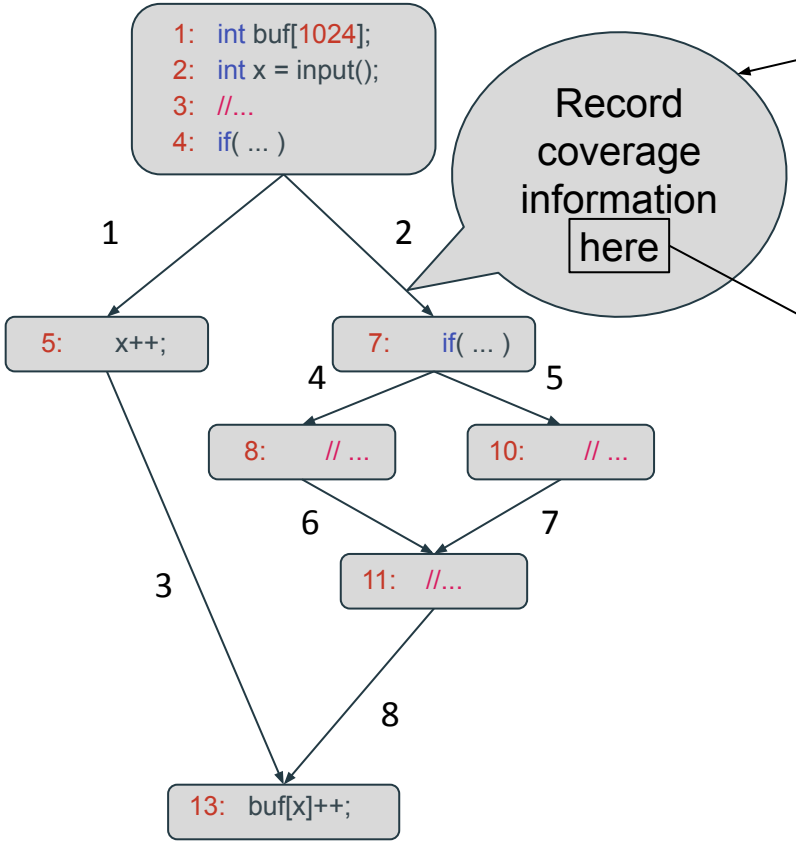


Compiler-level instrumentation

AFL & AFLGo


1: int buf[1024];
2: int x = input();
3: //...
4: if(...)
1
2
Record coverage information here
Compiler-level instrumentation
Coverage bitmap (shared memory)
5: x++;
7: if(...)
4
5
8: // ...
10: // ...
6
7
11: //...
3
8
13: buf[x]++;

AFL & AFLGo



Record coverage information here

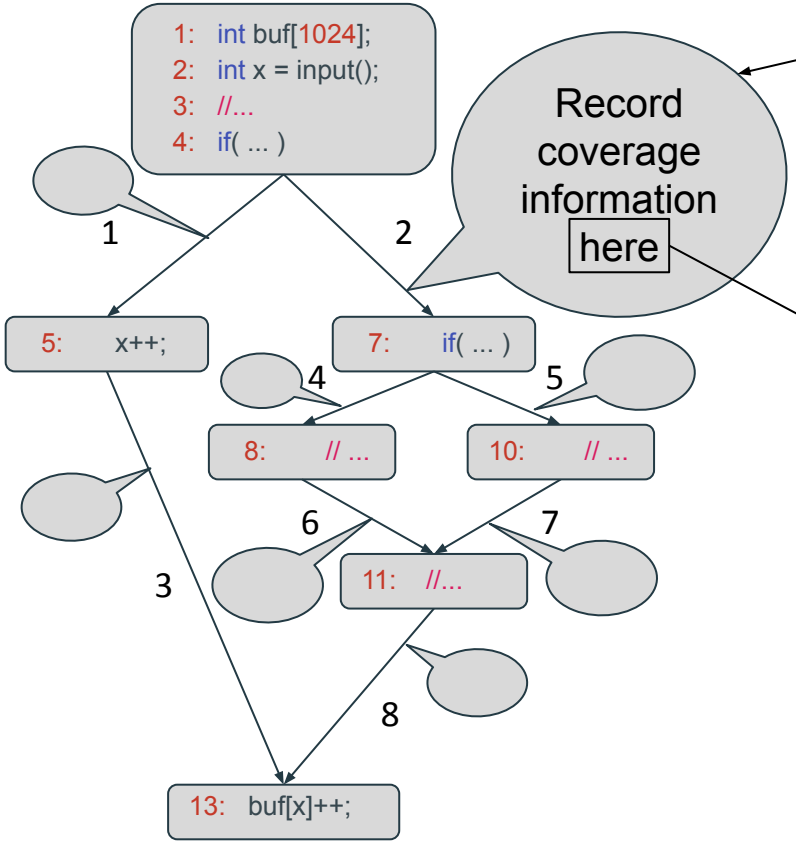
Compiler-level instrumentation

…

Fuzzer determines the interestingness with:

Coverage bitmap (shared memory)

AFL & AFLGo



Record coverage information here

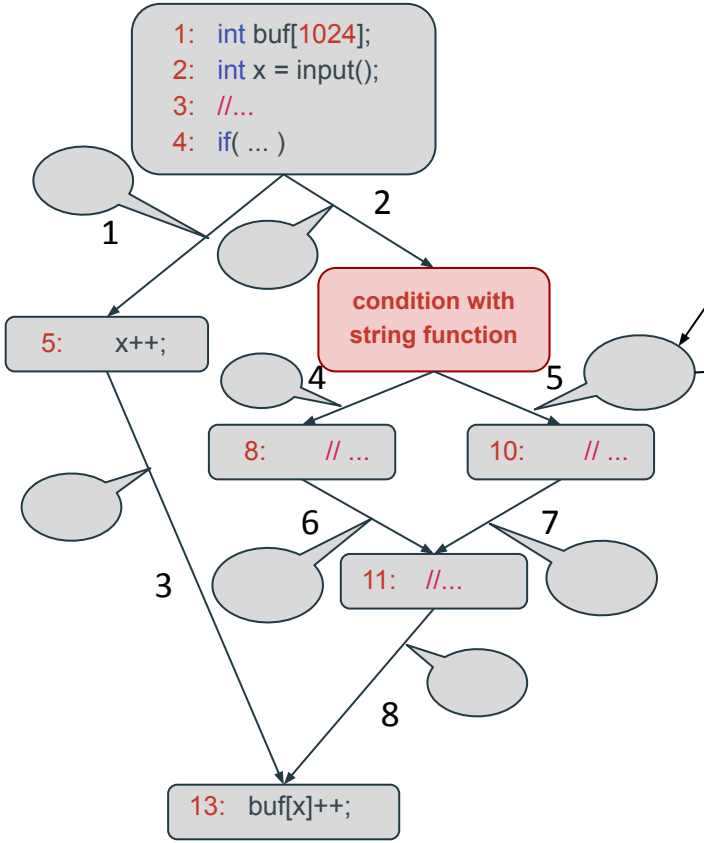
Compiler-level instrumentation

…

Fuzzer determines the interestingness with:

Coverage bitmap (shared memory)

smAFL



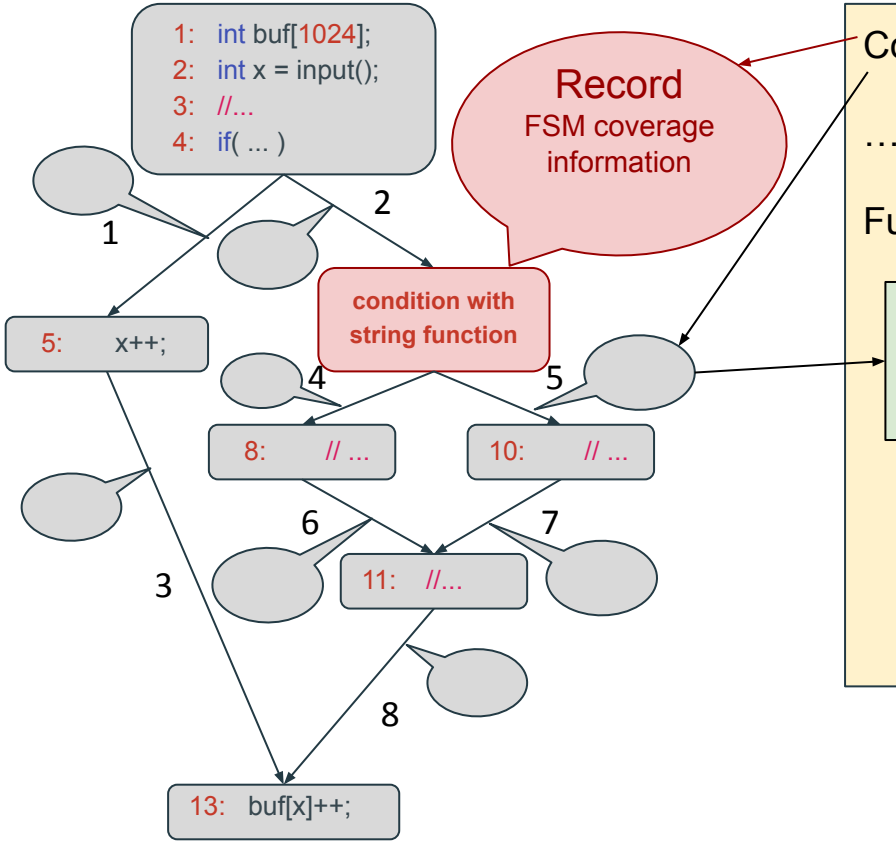
Compiler-level instrumentation

…

Fuzzer determines the interestingness with:

Coverage bitmap (shared memory)

smAFL



Record
FSM coverage information

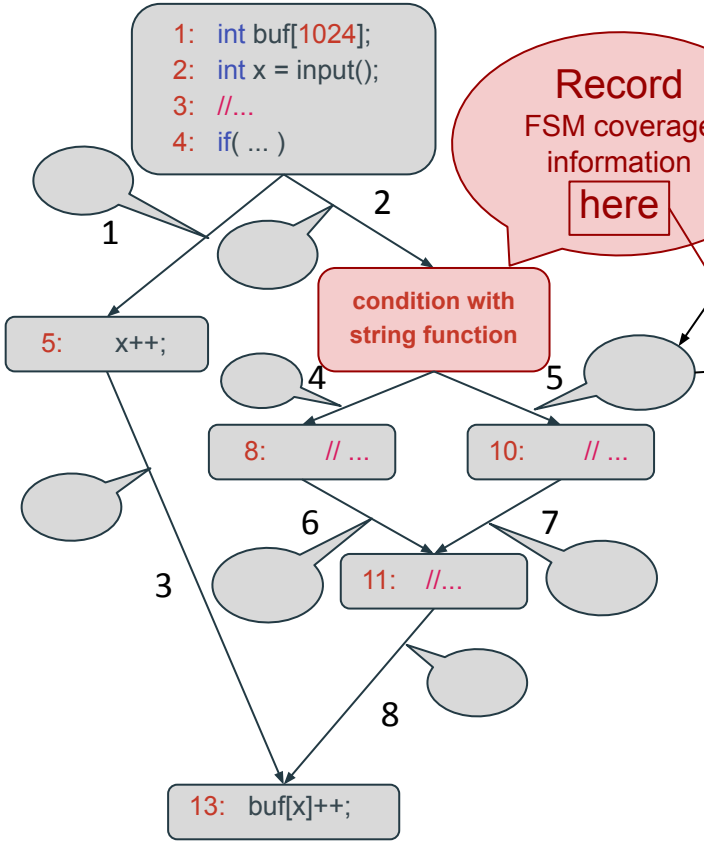
Compiler-level instrumentation

…

Fuzzer determines the interestingness with:

Coverage bitmap (shared memory)

smAFL



Record FSM coverage information here

Compiler-level instrumentation

…

Fuzzer determines the interestingness with:

Coverage bitmap (shared memory)

FSM Coverage Information (shared memory)

Evaluation

Evaluation: benchmark

CVE: Common Vulnerabilities and Exposures

=> We use four target CVEs from Binutils cxxfilt binary.

2016-4487, 2016-4489, 2016-4490, 2016-4492

Evaluation: benchmark

CVE: Common Vulnerabilities and Exposures

=> We use four target CVEs from Binutils cxxfilt binary.

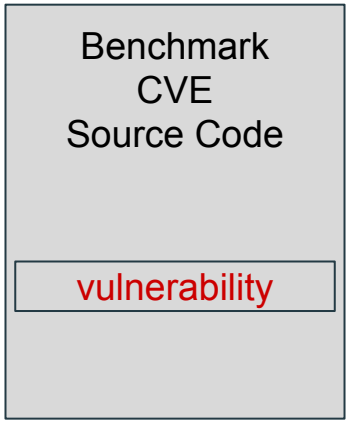
2016-4487, 2016-4489, 2016-4490, 2016-4492

Reason:

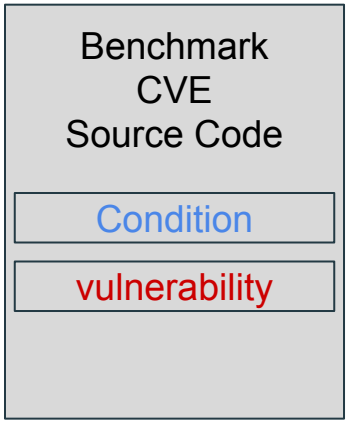
- Most commonly used targets in previous literature
- Receives string inputs
- Requires 500~1500 Sec. to reproduce

Evaluation: criteria

1. Original benchmark



2. Inject condition to guard the vulnerability



3. Run fuzzers and compare performance

Evaluation: setup

Designed blocking conditions that preserves the target CVE

Evaluation: setup

Designed blocking conditions that preserves the target CVE

EX) 2016-4487

Evaluation: setup

Designed blocking conditions that preserves the target CVE

EX) 2016-4487

Provided crashing input: _QA__1.

Evaluation: setup

Designed blocking conditions that preserves the target CVE

EX) 2016-4487

Provided crashing input: _QA__1.

Crashing behaviour preserved with any number in the input

Evaluation: setup

Designed blocking conditions that preserves the target CVE

EX) 2016-4487

Provided crashing input: _QA__1.

Crashing behaviour preserved with any number in the input

Insert conditions with string functions (i.e., strcmp(str, '123456'))

Evaluation: setup

Designed blocking conditions that preserves the target CVE

EX) 2016-4487

Provided crashing input: _QA__1.

Crashing behaviour preserved with any number in the input

Insert conditions with string functions (i.e., strcmp(str, '123456'))

Thus, limiting the crashing input to _QA__123456.

Evaluation: setup

Utilized three types of conditions: strcmp, strstr, atoi

12 targets: 4 original CVE targets * 3 types of conditions

Done & Todo

Done: Evaluation setup (baseline tools and targets)

Todo: Implementation

Summary

Goal: Improve Directed Greybox Fuzzing.

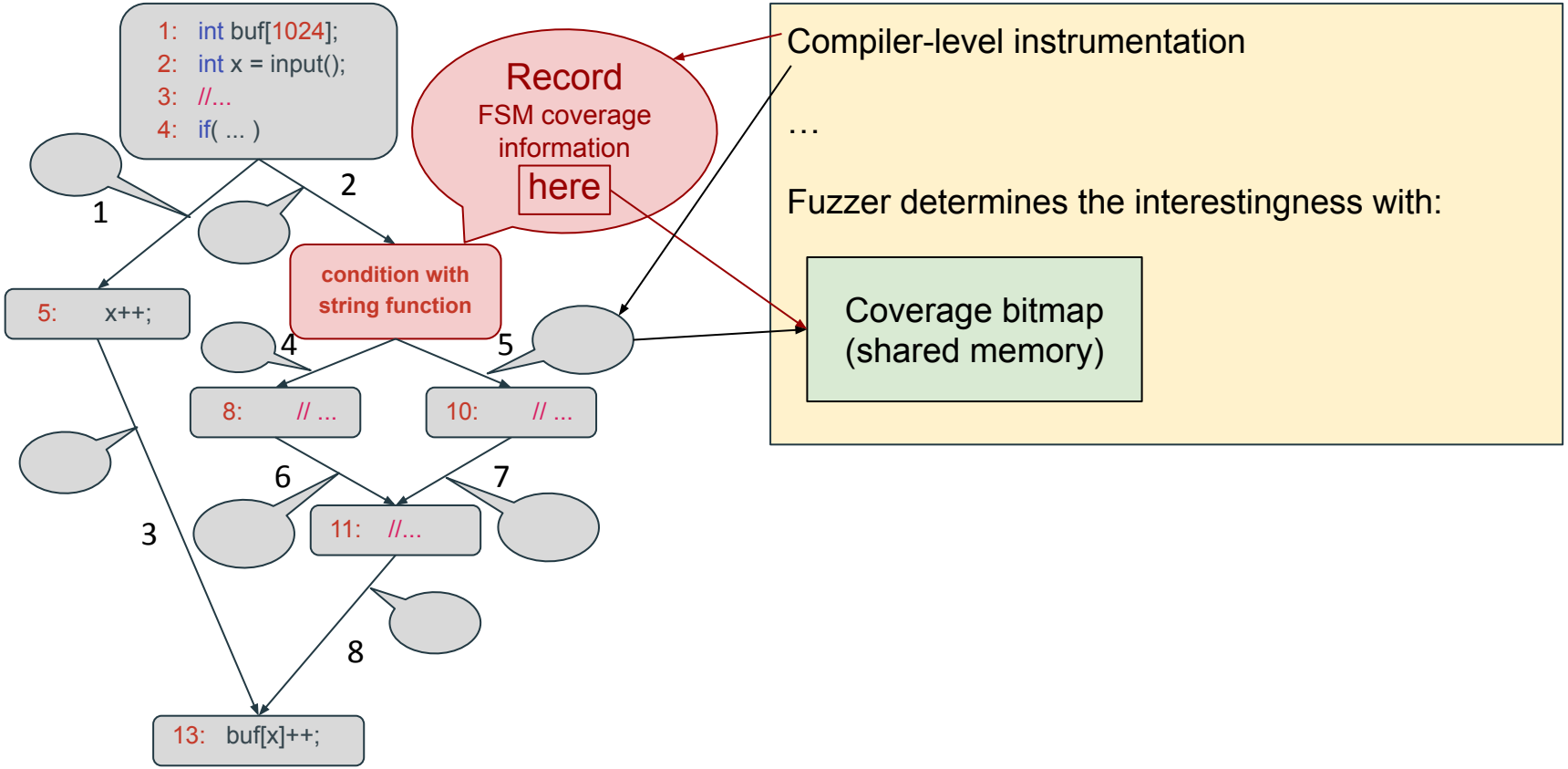
Problem: Greybox fuzzers struggle with the absence of code coverage.

Method: Use Finite State Machine to provide intermediate code coverage.

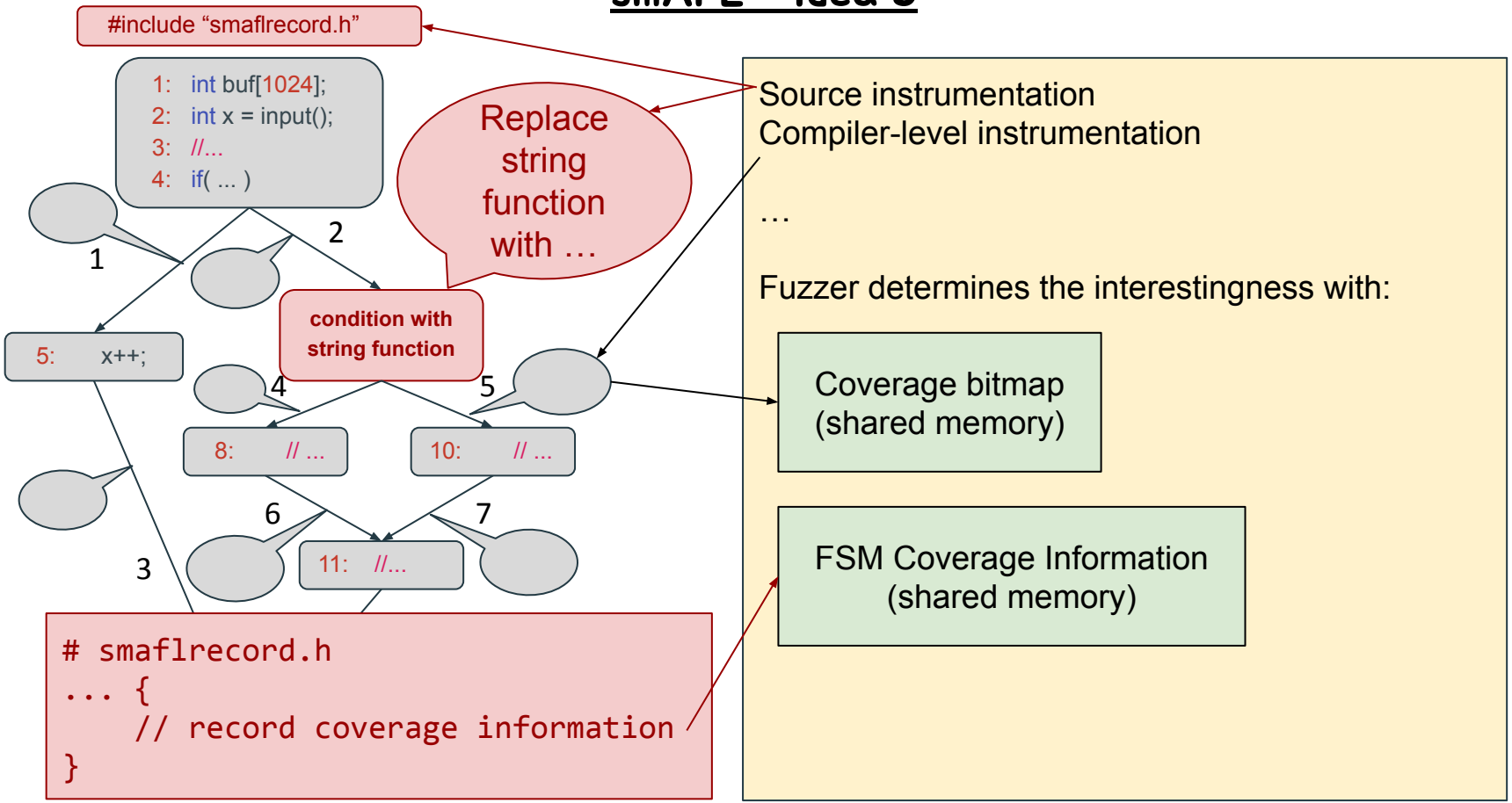
More detailed method: Utilize additional feedback from Finite State Machine coverage

Targets: 4 CVEs in cxxfilt * 3 types of blocking conditions

smAFL - Idea 1


1: int buf[1024];
2: int x = input();
3: //...
4: if(...)
1
2
Record
FSM coverage
information
here
condition with
string function
5: x++;
4
5
8: // ...
10: // ...
6
7
3
11: //...
8
13: buf[x]++;
Compiler-level instrumentation
…
Fuzzer determines the interestingness with:
Coverage bitmap
(shared memory)

smAFL - Idea 3


#include “smaflrecord.h”
1: int buf[1024];
2: int x = input();
3: //...
4: if(...)
Replace string function with …
1
2
condition with string function
5: x++;
4
5
8: // ...
10: // ...
6
7
3
11: //...
# smaflrecord.h
... {
// record coverage information
}
Source instrumentation
Compiler-level instrumentation
…
Fuzzer determines the interestingness with:
Coverage bitmap (shared memory)
FSM Coverage Information (shared memory)