UBFuzz:
Finding Bugs in Sanitizer Implementations

Steve Gustaman

Original paper by Shaohua Li and Zhendong Su
ASPLOS 2024

Motivation

- **Undefined behaviors (UB) are everywhere in software**

Motivation

- **Undefined behaviors (UB) are everywhere in software**
  - Buffer overflow
  - Integer overflow
  - Use after free

Motivation

- **Undefined behaviors (UB) are everywhere in software**
  - Buffer overflow
  - Integer overflow
  - Use after free

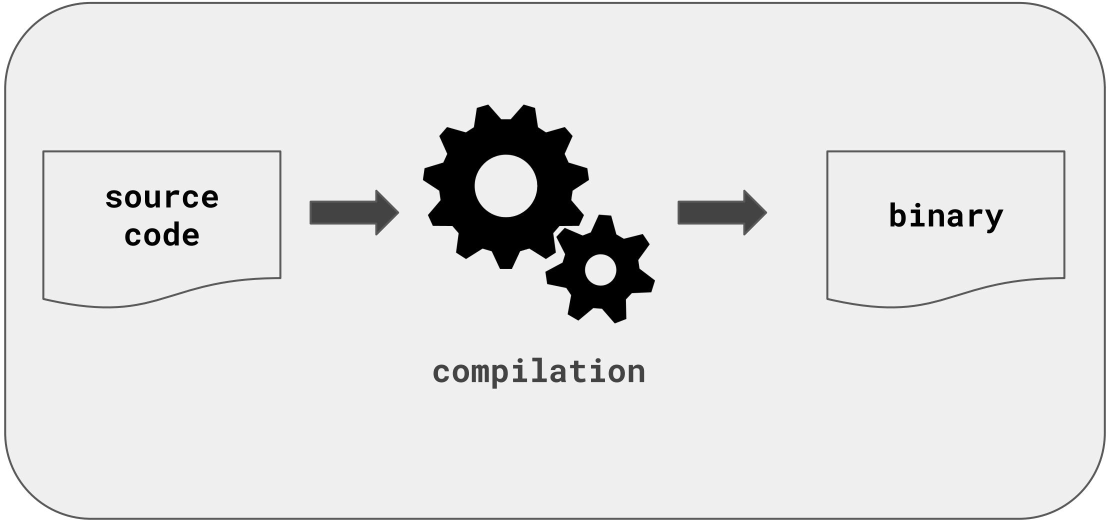
security issue!

Motivation

- Undefined behaviors (UB) are everywhere in software
- **Sanitizers are widely used to detect UB**

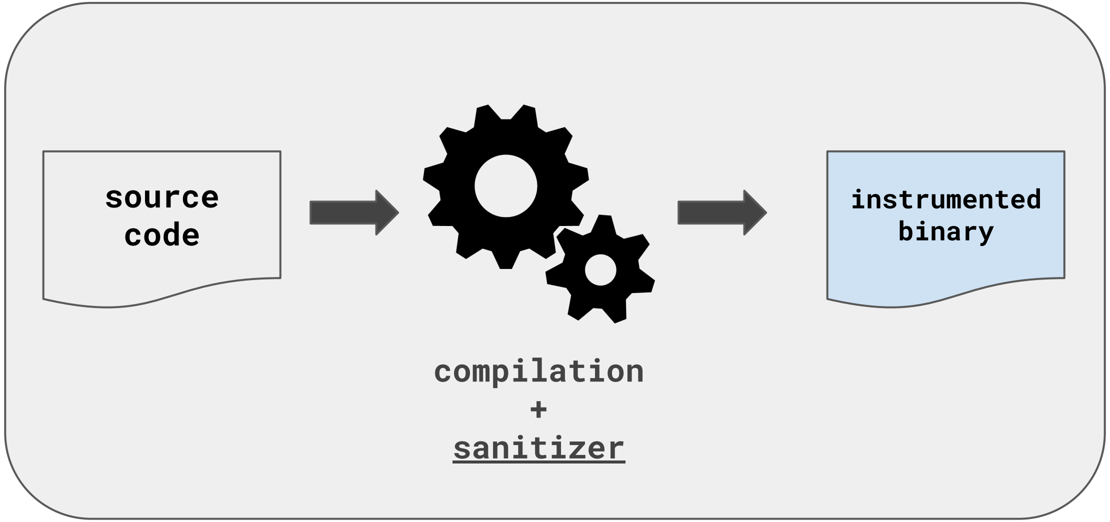
Motivation

- Undefined behaviors (UB) are everywhere in software
- **Sanitizers are widely used to detect UB**



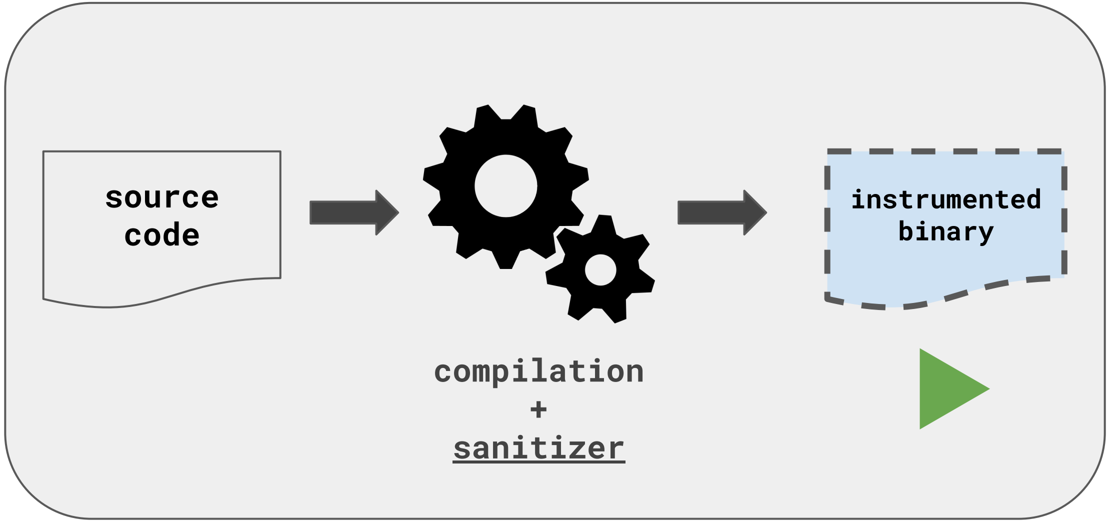
Motivation

- Undefined behaviors (UB) are everywhere in software
- **Sanitizers are widely used to detect UB**



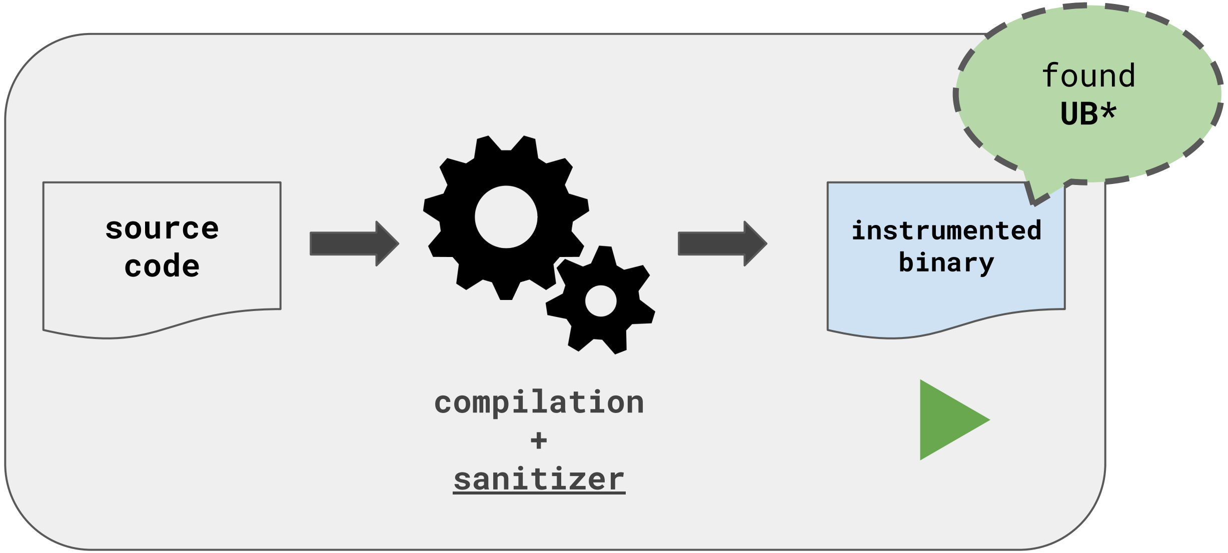
Motivation

- Undefined behaviors (UB) are everywhere in software
- **Sanitizers are widely used to detect UB**



Motivation

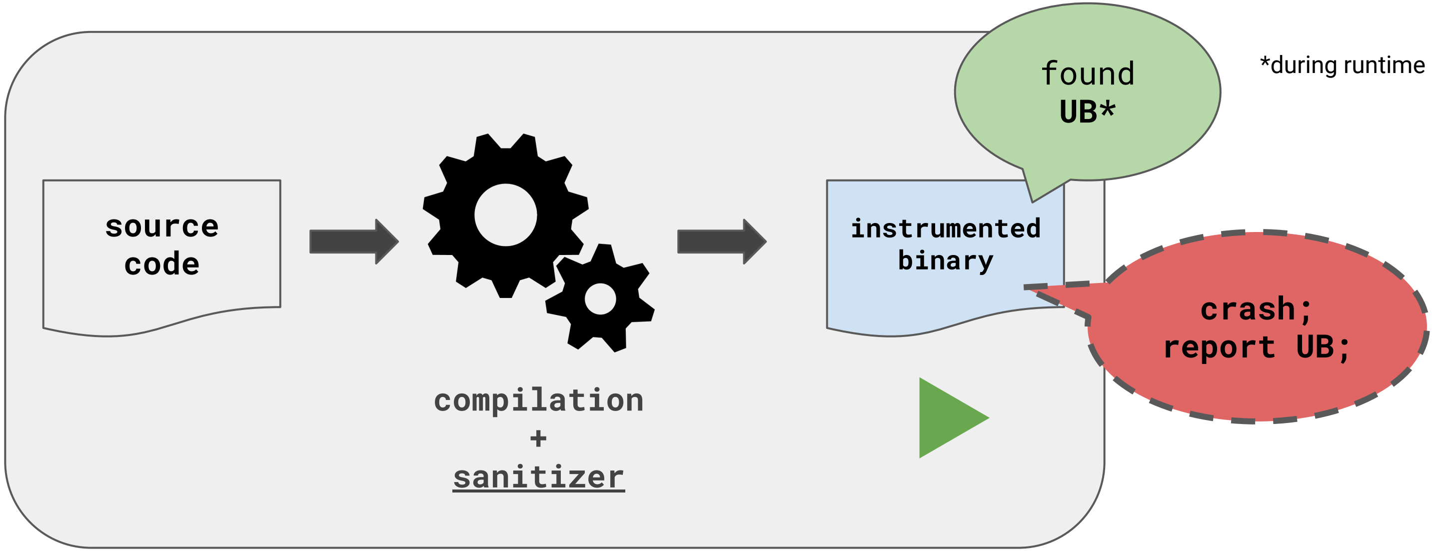
- Undefined behaviors (UB) are everywhere in software
- **Sanitizers are widely used to detect UB**



*during runtime

Motivation

- Undefined behaviors (UB) are everywhere in software
- **Sanitizers are widely used to detect UB**



Motivation

- Undefined behaviors (UB) are everywhere in software
- **Sanitizers are widely used to detect UB**

Motivation

- Undefined behaviors (UB) are everywhere in software
- **Sanitizers are widely used to detect UB**
  - **MSan:** uninitialized memory usage, ...
  - **UBSan:** integer overflow, ...
  - **ASan:** buffer overflow, ...

Motivation

- Undefined behaviors (UB) are everywhere in software
- **Sanitizers are widely used to detect UB**
  - **MSan:** uninitialized memory usage, ...
  - **UBSan:** integer overflow, ...
  - **ASan:** buffer overflow, ...

used as Fuzzer Oracle

Motivation

- Undefined behaviors (UB) are everywhere in software
- **Sanitizers are widely used to detect UB**
  - **MSan:** uninitialized memory usage, ...
  - **UBSan:** integer overflow, ...
  - **ASan:** buffer overflow, ...

used as Fuzzer Oracle

Google OSS-Fuzz reported >20K UBs

Motivation

- Undefined behaviors (UB) are everywhere in software
- Sanitizers are widely used to detect UB
- **Robustness and reliability of sanitizers are understudied**

Motivation

- Undefined behaviors (UB) are everywhere in software
- Sanitizers are widely used to detect UB
- **Robustness and reliability of sanitizers are understudied**
  - Last 5 years only 29 bug report in GCC and LLVM Sanitizers

Motivation

- Undefined behaviors (UB) are everywhere in software
- Sanitizers are widely used to detect UB
- **Robustness and reliability of sanitizers are understudied**
  - **Last 5 years** only 29 bug report in GCC and LLVM Sanitizers

Motivation

- Undefined behaviors (UB) are everywhere in software
- Sanitizers are widely used to detect UB
- **Robustness and reliability of sanitizers are understudied**
    - **Last 5 years** only **29 bug report** in GCC and LLVM Sanitizers

Motivation

- Undefined behaviors (UB) are everywhere in software
- Sanitizers are widely used to detect UB
- **Robustness and reliability of sanitizers are understudied**
  - Last 5 years only 29 bug report in GCC and LLVM Sanitizers
  - Out of 29, **66% are FP errors**

Motivation

- Undefined behaviors (UB) are everywhere in software
- Sanitizers are widely used to detect UB
- **Robustness and reliability of sanitizers are <u>understudied</u>**
    - Last 5 years only 29 bug report in GCC and LLVM Sanitizers
    - Out of 29, 66% are FP errors
    - The rest, **<u>34% are FN errors</u>**

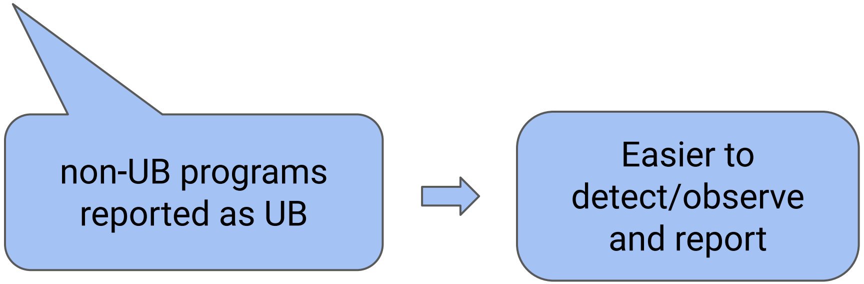
Motivation

- Undefined behaviors (UB) are everywhere in software
- Sanitizers are widely used to detect UB
- **Robustness and reliability of sanitizers are understudied**
  - Last 5 years only 29 bug report in GCC and LLVM Sanitizers
  - Out of 29, **66% are FP errors**
  - The rest, 34% are FN errors

non-UB programs
reported as UB

Motivation

- Undefined behaviors (UB) are everywhere in software
- Sanitizers are widely used to detect UB
- **Robustness and reliability of sanitizers are <u>understudied</u>**
    - Last 5 years only 29 bug report in GCC and LLVM Sanitizers
    - Out of 29, **<u>66% are FP errors</u>**
    - The rest, 34% are FN errors



Motivation

- Undefined behaviors (UB) are everywhere in software
- Sanitizers are widely used to detect UB
- **Robustness and reliability of sanitizers are <u>understudied</u>**
    - Last 5 years only 29 bug report in GCC and LLVM Sanitizers
    - Out of 29, **<u>66% are FP errors</u>**
    - The rest, 34% are FN errors

non-UB programs **<u>reported as UB</u>** → **<u>Easier</u>** to detect/observe and report

Motivation

- Undefined behaviors (UB) are everywhere in software
- Sanitizers are widely used to detect UB
- **Robustness and reliability of sanitizers are understudied**
  - Last 5 years only 29 bug report in GCC and LLVM Sanitizers
  - Out of 29, 66% are FP errors
  - The rest, **34% are FN errors**

missing,
unreported UB

Motivation

- Undefined behaviors (UB) are everywhere in software
- Sanitizers are widely used to detect UB
- **Robustness and reliability of sanitizers are <u>understudied</u>**
  - Last 5 years only 29 bug report in GCC and LLVM Sanitizers
  - Out of 29, 66% are FP errors
  - The rest, **<u>34% are FN errors</u>**

missing,
<u>unreported UB</u>

→

<u>Hard</u> to
detect/observe
and report

Motivation

- Undefined behaviors (UB) are everywhere in software
- Sanitizers are widely used to detect UB
- **Robustness and reliability of sanitizers are understudied**
  - Last 5 years only 29 bug report in GCC and LLVM Sanitizers
  - Out of 29, 66% are FP errors
  - The rest, **34% are FN errors**

missing, unreported UB → Hard to detect/observe and report

possibly much more **hidden FN errors**

Motivation

- Undefined behaviors (UB) are everywhere in software
- Sanitizers are widely used to detect UB
- Robustness and reliability of sanitizers are understudied
- **It is an important problem to find FN bugs in sanitizers**

Motivation

- Undefined behaviors (UB) are everywhere in software
- Sanitizers are widely used to detect UB
- Robustness and reliability of sanitizers are understudied
- **It is an important problem to find FN bugs in sanitizers**

missing,
unreported UB

Motivation

- Undefined behaviors (UB) are everywhere in software
- Sanitizers are widely used to detect UB
- Robustness and reliability of sanitizers are understudied
- **It is an important problem to find FN bugs in sanitizers**

missing,
unreported UB

Sanitizer: **many FNs = ineffective**

Motivation

- Undefined behaviors (UB) are everywhere in software
- Sanitizers are widely used to detect UB
- Robustnes
- It is an im

but how to effectively find **FN bugs** in sanitizers?

UBFuzz:

Finding Bugs in Sanitizer Implementations

with shadow statement insertion based program generation
and crash-site mapping oracle

UBFuzz Key Idea

- **Goal**: Find FN bugs in sanitizer (undetected, but actual UB)

UBFuzz Key Idea

- **Goal**: Find FN bugs in sanitizer (undetected, but actual UB)
  - Generate program with UB

UBFuzz Key Idea

- **Goal**: Find FN bugs in sanitizer (undetected, but actual UB)
  - Generate program with UB
  - Check if sanitizer is able to detect the UB

UBFuzz Key Idea

- **Goal**: Find FN bugs in sanitizer (undetected, but actual UB)
  - Generate program with UB
  - Check if sanitizer is able to detect the UB
    - **Cannot detect → sanitizer FN bug**

UB Program Generation

UB Program Generation

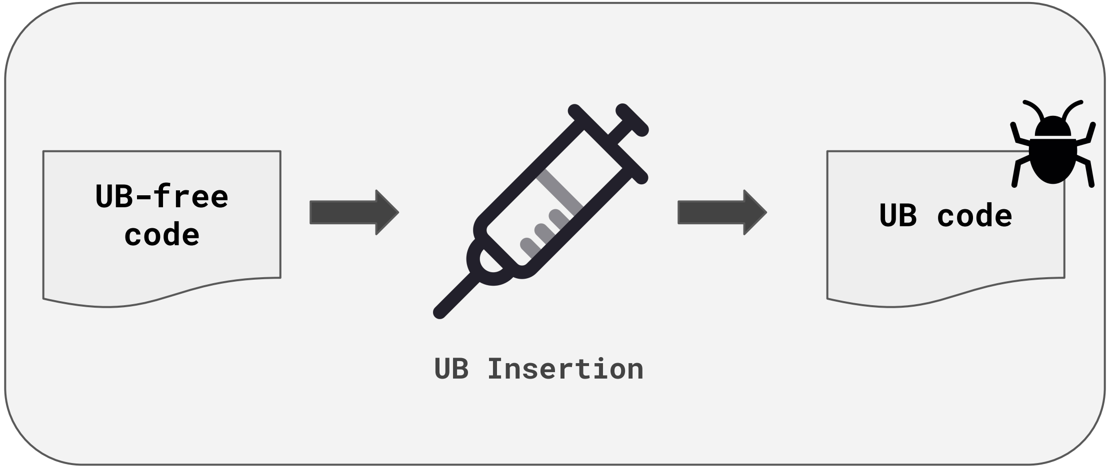
- UB-free program generation: **CSmith**

UB Program Generation

- UB-free program generation: **CSmith**
- Idea: Introduce UB to generated UB-free program

UB Program Generation

- UB-free program generation: **CSmith**
- Idea: Introduce UB to generated UB-free program



UB Program Generation


```
struct a { int x };
struct a b[2];
struct a *c = b, *d = b;

int main() {

  *c = *b;


  *c = *(d);
  return c->x;
}
```

*example for stack buffer overflow UB insertion

UB Program Generation


```
struct a { int x };
struct a b[2];
struct a *c = b, *d = b;

int main() {

  *c = *b;


  *c = *(d);
  return c->x;
}
```

*example for stack buffer overflow UB insertion

UB Program Generation

1. **Identify and profile target**

```
struct a { int x };
struct a b[2];
struct a *c = b, *d = b;

int main() {

  *c = *b;


  *c = *(d);
  return c->x;
}
```

*example for stack buffer overflow UB insertion

UB Program Generation

1. **Identify and profile target**
    - stack buffers

```c
struct a { int x };
struct a b[2];
struct a *c = b, *d = b;

int main() {

  *c = *b;


  *c = *(d);
  return c->x;
}
```

*example for stack buffer overflow UB insertion

UB Program Generation

1. **Identify and profile target**
   - stack buffers

```
struct a { int x };
struct a b[2];
struct a *c = b, *d = b;

int main() {

  *c = *b;


  *c = *(d);
  return c->x;
}
```

UB Program Generation

*example for stack buffer overflow UB insertion

1. **Identify and profile target**
   - stack buffers

```
struct a { int x };
struct a b[2];
struct a *c = b, *d = b;

int main() {
  LOG_BufRange(&b[0], sizeof(b));
  *c = *b;


  *c = *(d);
  return c->x;
}
```

*example for stack buffer overflow UB insertion

UB Program Generation

1. **Identify and profile target**
   - stack buffers
   - memory accesses

```
struct a { int x };
struct a b[2];
struct a *c = b, *d = b;

int main() {
  LOG_BufRange(&b[0], sizeof(b));
  *c = *b;


  *c = *(d);
  return c->x;
}
```

UB Program Generation

*example for stack buffer overflow UB insertion

1. **Identify and profile target**
    - stack buffers
    - memory accesses

```
struct a { int x };
struct a b[2];
struct a *c = b, *d = b;

int main() {
  LOG_BufRange(&b[0], sizeof(b));
  *c = *b;

  LOG_BufAccess(d);
  *c = *(d);
  return c->x;
}
```

*example for stack buffer overflow UB insertion

UB Program Generation

1. Identify and profile target
   - stack buffers
   - memory accesses
2. **Compile and execute to obtain runtime information**

```
struct a { int x };
struct a b[2];
struct a *c = b, *d = b;

int main() {
  LOG_BufRange(&b[0], sizeof(b));
  *c = *b;

  LOG_BufAccess(d);
  *c = *(d);
  return c->x;
}
```

*example for stack buffer overflow UB insertion

UB Program Generation

1. Identify and profile target
   - stack buffers
   - memory accesses
2. **Compile and execute to obtain runtime information**

```
struct a { int x };
struct a b[2];
struct a *c = b, *d = b;

int main() {
  LOG_BufRange(&b[0], sizeof(b));
  *c = *b;

  LOG_BufAccess(d);
  *c = *(d);
  return c->x;
}
```

*example for stack buffer overflow UB insertion

UB Program Generation

1. Identify and profile target
   - stack buffers
   - memory accesses
2. **Compile and execute to obtain runtime information**

```
struct a { int x };
struct a b[2];
struct a *c = b, *d = b;

int main() {
  LOG_BufRange(&b[0], sizeof(b));
  *c = *b;

  LOG_BufAccess(d);
  *c = *(d);
  return c->x;
}
```

range:
[0x1230, 0x1238]

UB Program Generation

*example for stack buffer overflow UB insertion

1. Identify and profile target
   - stack buffers
   - memory accesses
2. **Compile and execute to obtain runtime information**

```c
struct a { int x };
struct a b[2];
struct a *c = b, *d = b;

int main() {
  LOG_BufRange(&b[0], sizeof(b));
  *c = *b;

  LOG_BufAccess(d);
  *c = *(d);
  return c->x;
}
```

range:
[0x1230, 0x1238]

access:
[0x1230, 0x1234]

*example for stack buffer overflow UB insertion

UB Program Generation

1. Identify and profile target
   - stack buffers
   - memory accesses
2. Compile and execute to obtain runtime information
3. **Introduce UB**

```
struct a { int x };
struct a b[2];
struct a *c = b, *d = b;

int main() {
  LOG_BufRange(&b[0], sizeof(b));
  *c = *b;

  LOG_BufAccess(d);
  *c = *(d);
  return c->x;
}
```

range:
[0x1230, 0x1238]

access:
[0x1230, 0x1234]

UB Program Generation

*example for stack buffer overflow UB insertion

1. Identify and profile target
   - stack buffers
   - memory accesses
2. Compile and execute to obtain runtime information
3. **Introduce UB**

```
struct a { int x };
struct a b[2];
struct a *c = b, *d = b;

int main() {
  LOG_BufRange(&b[0], sizeof(b));
  *c = *b;

  LOG_BufAccess(d);
  *c = *(d);
  return c->x;
}
```

range:
[0x1230, 0x1238]

access:
[0x1230, 0x1234]

UB Program Generation

*example for stack buffer overflow UB insertion

1. Identify and profile target
    - stack buffers
    - memory accesses
2. Compile and execute to obtain runtime information
3. **Introduce UB**

```
struct a { int x };
struct a b[2];
struct a *c = b, *d = b;
int k = 0;
int main() {
  LOG_BufRange(&b[0], sizeof(b));
  *c = *b;
  k = 2;
  LOG_BufAccess(d);
  *c = *(d+k);
  return c->x;
}
```

range:
[0x1230, 0x1238]

access:
[0x1230, 0x1234]

*example for stack buffer overflow UB insertion

UB Program Generation

1. Identify and profile target
   - stack buffers
   - memory accesses
2. Compile and execute to obtain runtime information
3. Introduce UB
4. **Remove profiling**

```
struct a { int x };
struct a b[2];
struct a *c = b, *d = b;
int k = 0;
int main() {
  LOG_BufRange(&b[0], sizeof(b));
  *c = *b;
  k = 2;
  LOG_BufAccess(d);
  *c = *(d+k);
  return c->x;
}
```

range:
[0x1230, 0x1238]

access:
[0x1230, 0x1234]

*example for stack buffer overflow UB insertion

UB Program Generation

1. Identify and profile target
   - stack buffers
   - memory accesses
2. Compile and execute to obtain runtime information
3. Introduce UB
4. **Remove profiling**

```
struct a { int x };
struct a b[2];
struct a *c = b, *d = b;
int k = 0;
int main() {

  *c = *b;
  k = 2;

  *c = *(d+k);
  return c->x;
}
```

*example for stack buffer overflow UB insertion

UB Program Generation

1. Identify and profile target
    - stack buffers
    - memory accesses
2. Compile and execute to obtain runtime information
3. Introduce UB
4. **Remove profiling**

```
struct a { int x };
struct a b[2];
struct a *c = b, *d = b;
int k = 0;
int main() {

  *c = *b;
  k = 2;

  *c = *(d+k);
  return c->x;
}
```

*example for stack buffer overflow UB insertion

UB Program Generation

1. Identify and profile target
   - stack buffers
   - memory accesses
2. Compile and execute to obtain runtime information
3. Introduce UB
4. **Remove profiling**

```
struct a { int x };
struct a b[2];
struct a *c = b, *d = b;
int k = 0;
int main() {

  *c = *b;
  k = 2;

  *c = *(d+k);
  return c->x;
}
```

shadow statement insertion

UB Program Generation

*example for stack buffer overflow UB insertion

1. Identify and profile target
   - stack buffers
   - memory accesses
2. Compile and execute to obtain runtime information
3. Introduce UB
4. Remove profiling

```c
struct a { int x };
struct a b[2];
struct a *c = b, *d = b;
int k = 0;
int main() {

  *c = *b;
  k = 2;

  *c = *(d+k);
  return c->x;
}
```

UB Program Generation

*example for stack buffer overflow UB insertion

1. Identify and profile target
   - stack buffers
   - memory accesses
2. Compile and execute to obtain runtime information
3. Introduce UB
4. Remove profiling

```
struct a { int x };
struct a b[2];
struct a *c = b, *d = b;
int k = 0;
int main() {

  *c = *b;
  k = 2;

  *c = *(d+k);
  return c->x;
}
```

- Approach is general
- Applied to 9 UB types in UBFuzz

UBFuzz Key Idea

- **Goal**: Find FN bugs in sanitizer (undetected, but actual UB)
  - Generate program with UB
  - Check if sanitizer is able to detect the UB
    - Cannot detect → **sanitizer FN bug**

UBFuzz Key Idea

- **Goal**: Find FN bugs in sanitizer (undetected, but actual UB)
    - **Generate program with UB ✔**
    - Check if sanitizer is able to detect the UB
        - Cannot detect → **sanitizer FN bug**

UB Test Oracle with Crash-site Mapping

UB Test Oracle with Crash-site Mapping

- **Can’t detect UB from generated program w/ UB → FN bug**

UB Test Oracle with Crash-site Mapping

- Can’t detect UB from generated program w/ UB → FN bug
- **Compiler optimization may optimize UB-inducing code**

UB Test Oracle with Crash-site Mapping

- Can’t detect UB from generated program w/ UB → FN bug
- **Compiler optimization may optimize UB-inducing code**
  - Unreported UB <u>not always sanitizer FN bug</u>

UB Test Oracle with Crash-site Mapping

- ~~Can't detect UB from generated program w/ UB → FN bug~~
- **Compiler optimization may optimize UB-inducing code**
  - Unreported UB not always sanitizer FN bug

UB Test Oracle with Crash-site Mapping

- ~~Can't detect UB from generated program w/ UB → FN bug~~
- Compiler optimization may optimize UB-inducing code
  - Unreported UB <u>not always sanitizer FN bug</u>
- **Testing only -O0 is incomplete**

UB Test Oracle with Crash-site Mapping

- ~~Can’t detect UB from generated program w/ UB → FN bug~~
- Compiler optimization may optimize UB-inducing code
  - Unreported UB <u>not always sanitizer FN bug</u>
- Testing only -O0 is incomplete
- **Differential testing with 2 compilers**

UB Test Oracle with Crash-site Mapping

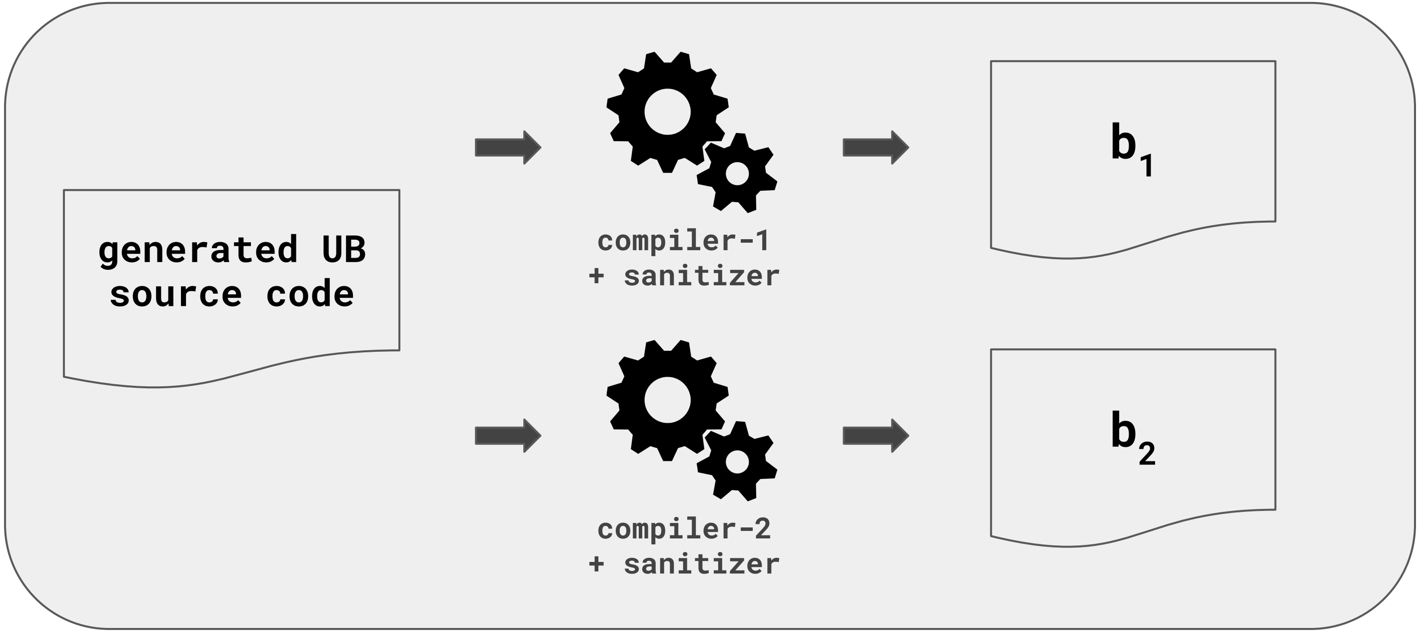
- Differential testing with 2 compilers

UB Test Oracle with Crash-site Mapping

- Differential testing with 2 compilers
    - e.g. **`gcc ASAN -O0`** and **`clang ASAN -O3`**

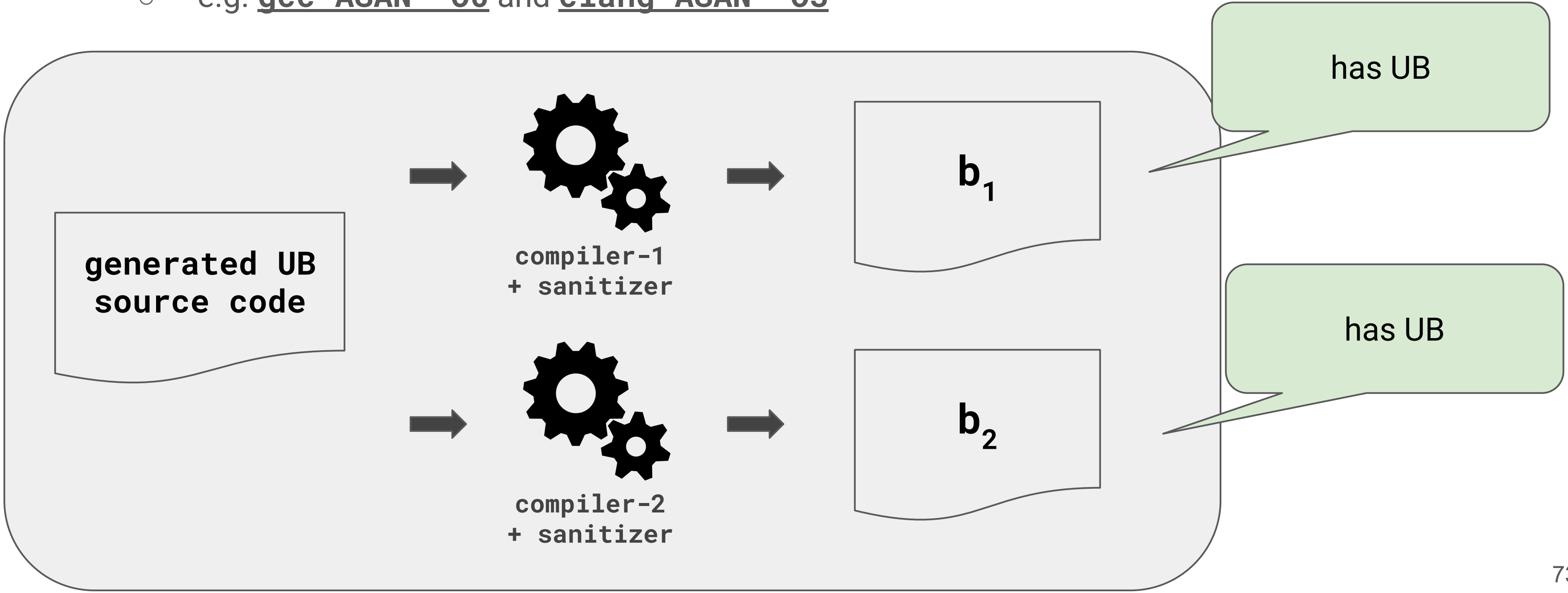
UB Test Oracle with Crash-site Mapping

- Differential testing with 2 compilers
  - e.g. `gcc ASAN -O0` and `clang ASAN -O3`



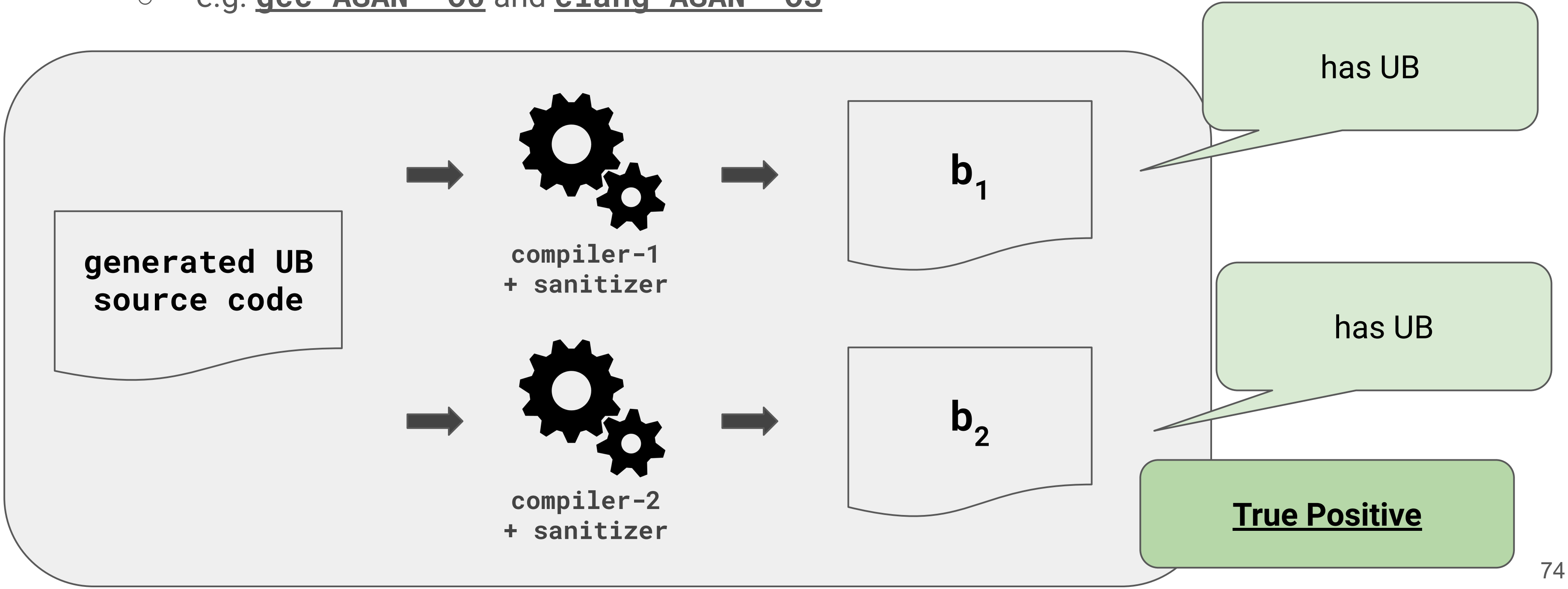
UB Test Oracle with Crash-site Mapping

- Differential testing with 2 compilers
  - e.g. **`gcc ASAN -O0`** and **`clang ASAN -O3`**



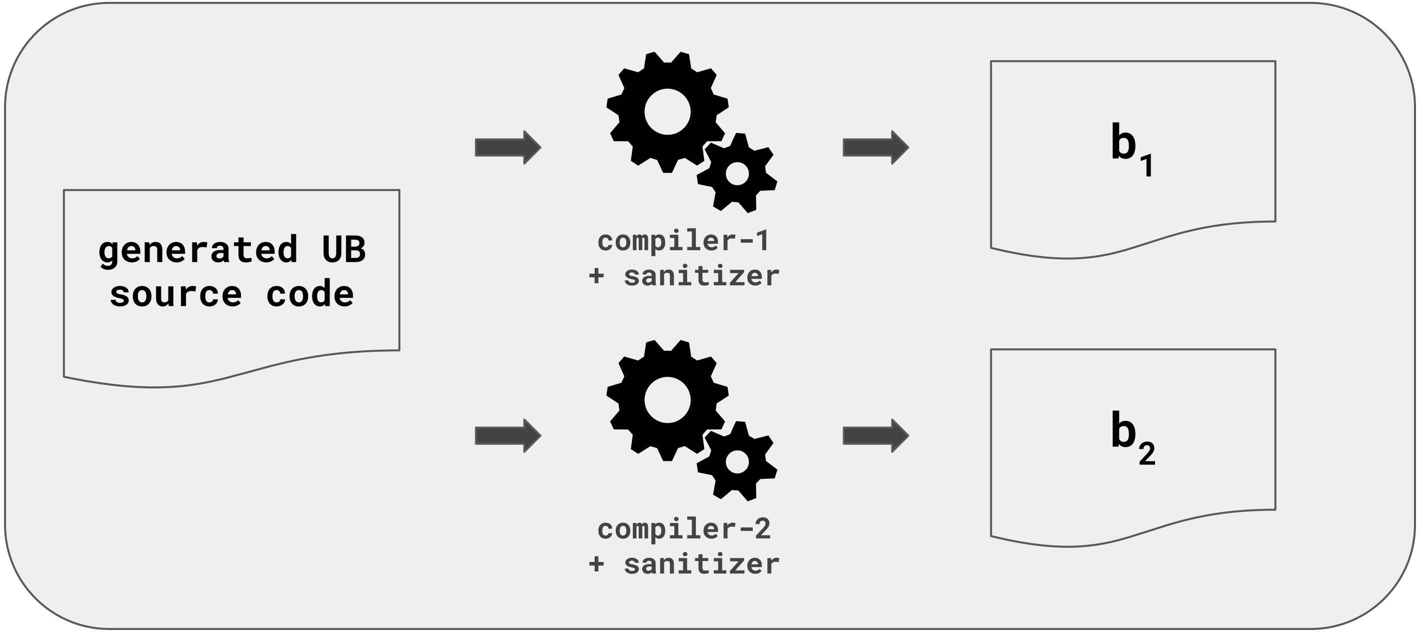
UB Test Oracle with Crash-site Mapping

- Differential testing with 2 compilers
  - e.g. **`gcc ASAN -O0`** and **`clang ASAN -O3`**



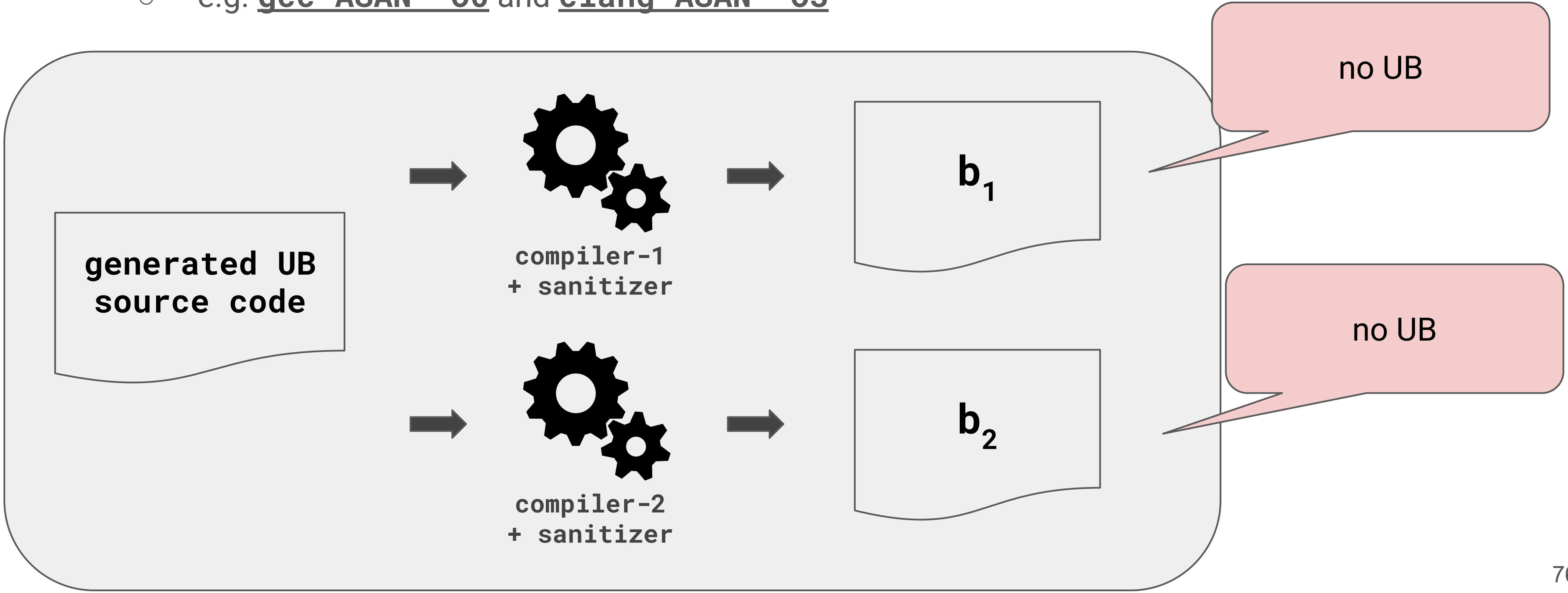
UB Test Oracle with Crash-site Mapping

- Differential testing with 2 compilers
  - e.g. **`gcc ASAN -O0`** and **`clang ASAN -O3`**



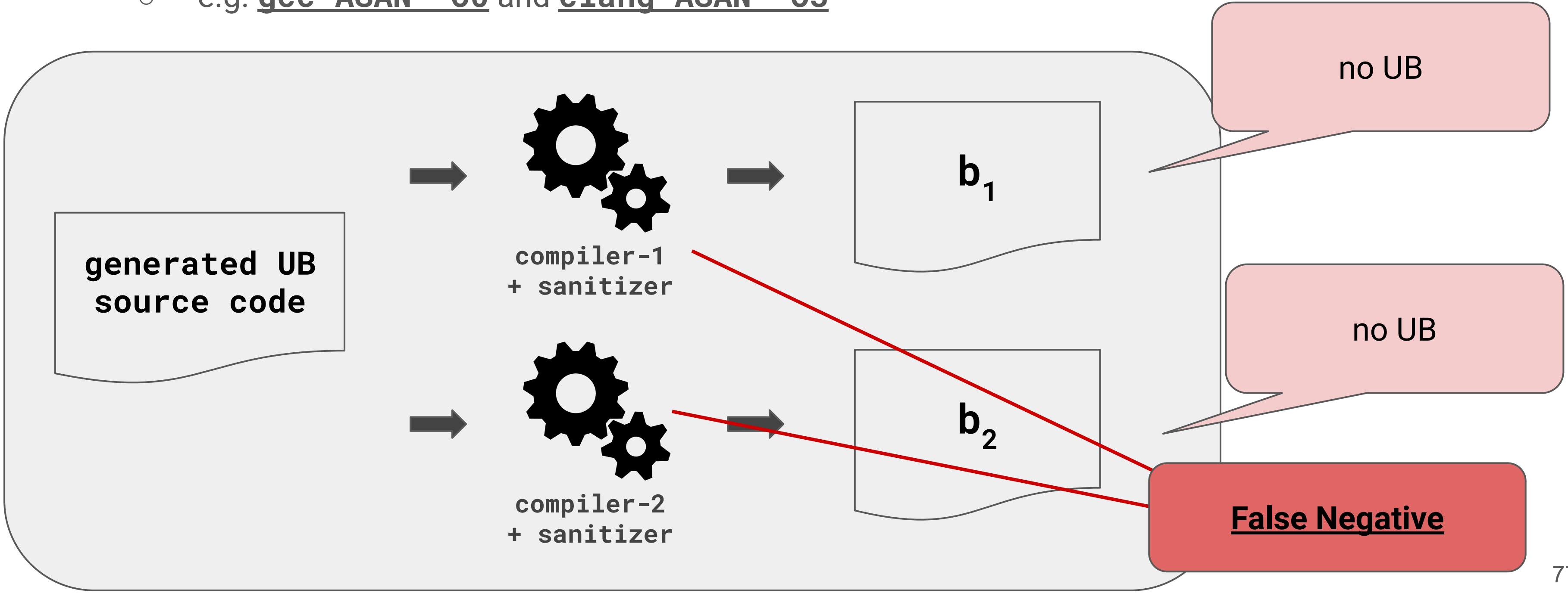
UB Test Oracle with Crash-site Mapping

- Differential testing with 2 compilers
  - e.g. **`gcc ASAN -O0`** and **`clang ASAN -O3`**



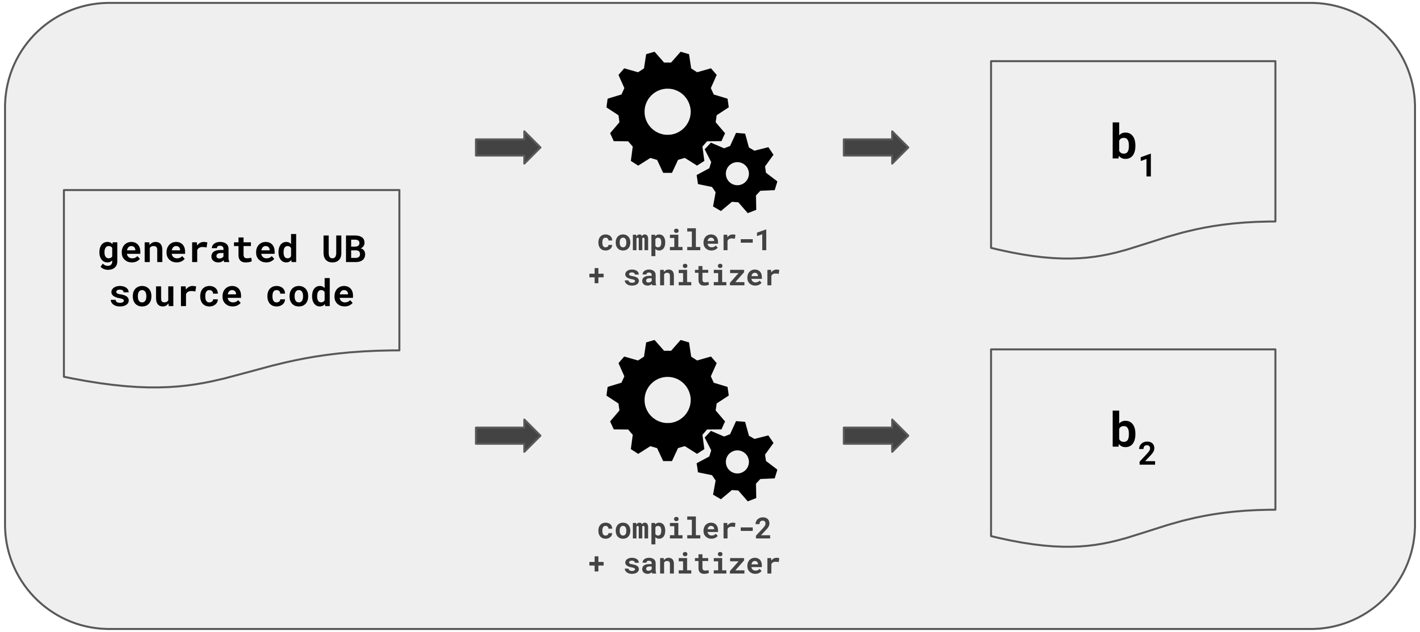
UB Test Oracle with Crash-site Mapping

- Differential testing with 2 compilers
  - e.g. **gcc ASAN -O0** and **clang ASAN -O3**



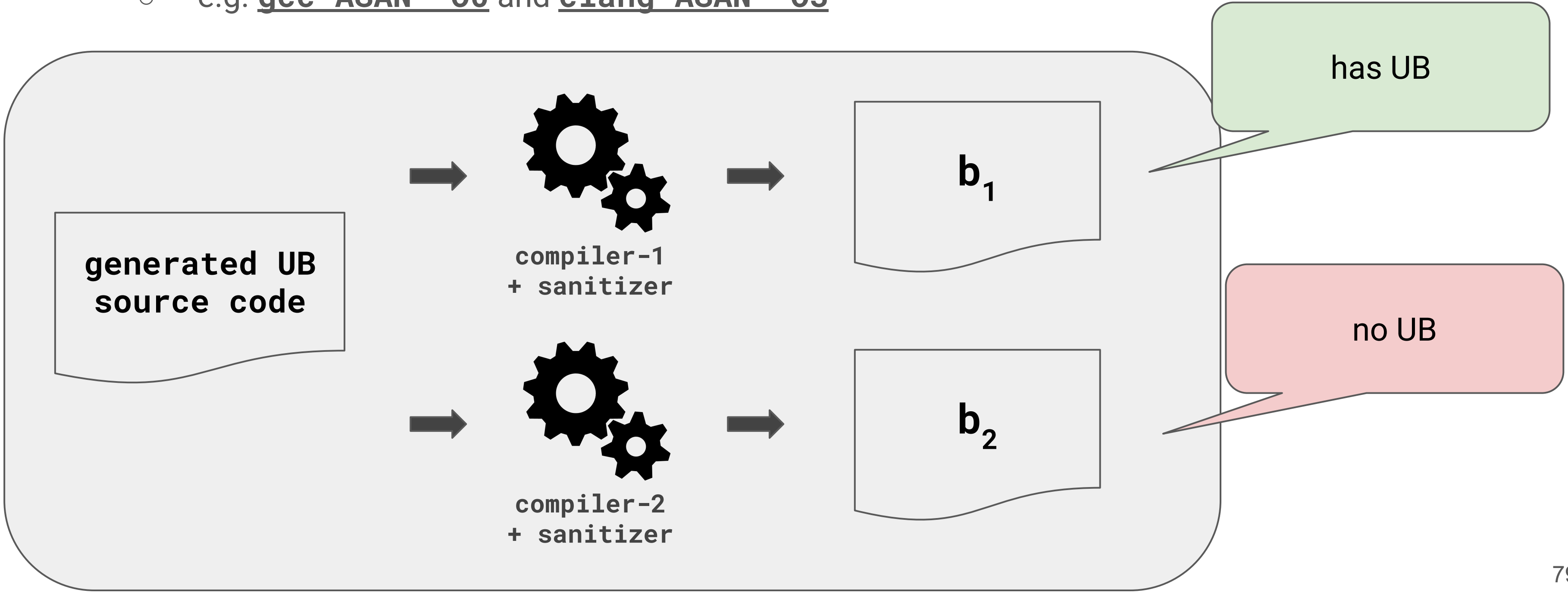
UB Test Oracle with Crash-site Mapping

- Differential testing with 2 compilers
  - e.g. `gcc ASAN -O0` and `clang ASAN -O3`



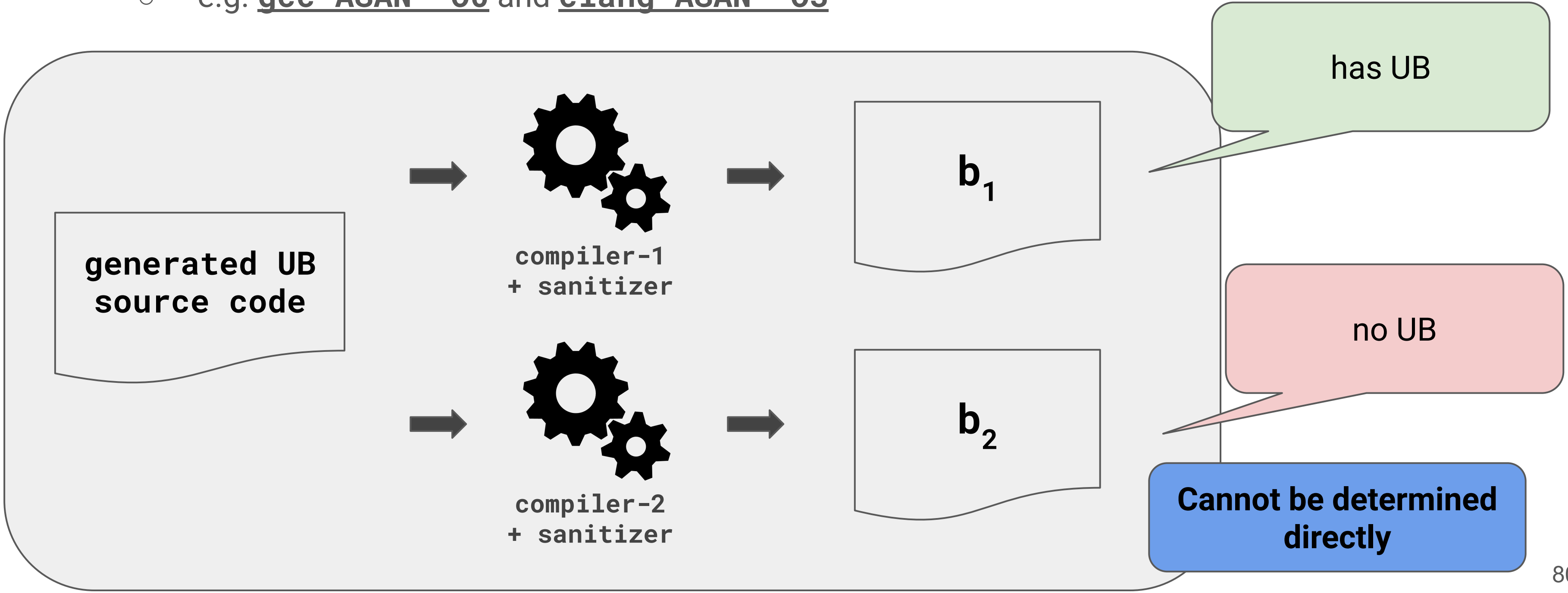
UB Test Oracle with Crash-site Mapping

- Differential testing with 2 compilers
  - e.g. **`gcc ASAN -O0`** and **`clang ASAN -O3`**

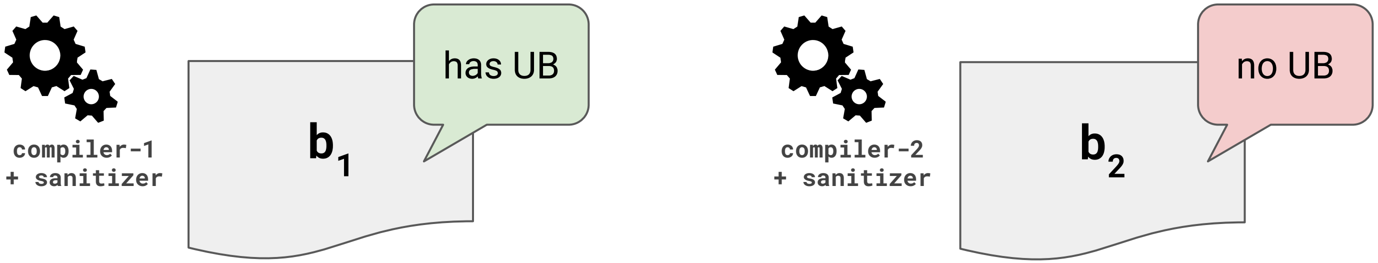


UB Test Oracle with Crash-site Mapping

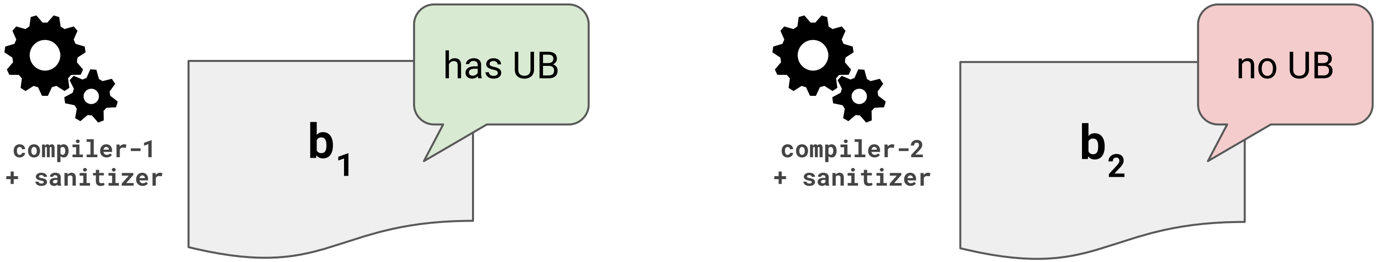
- Differential testing with 2 compilers
  - e.g. **gcc ASAN -O0** and **clang ASAN -O3**



UB Test Oracle with Crash-site Mapping

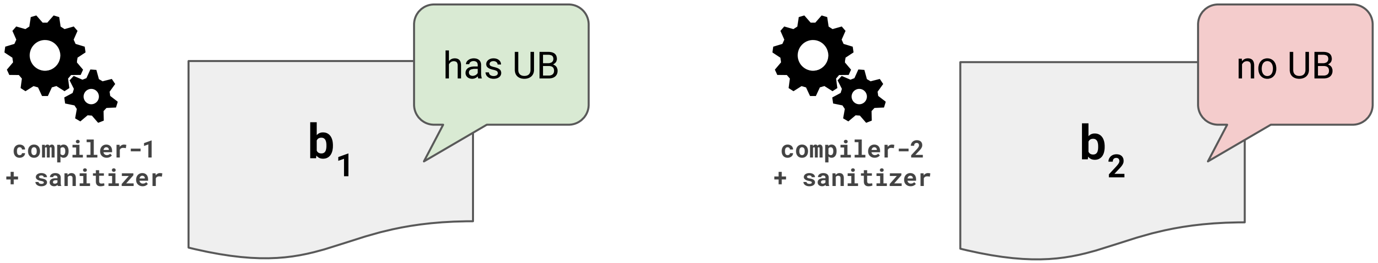


UB Test Oracle with Crash-site Mapping



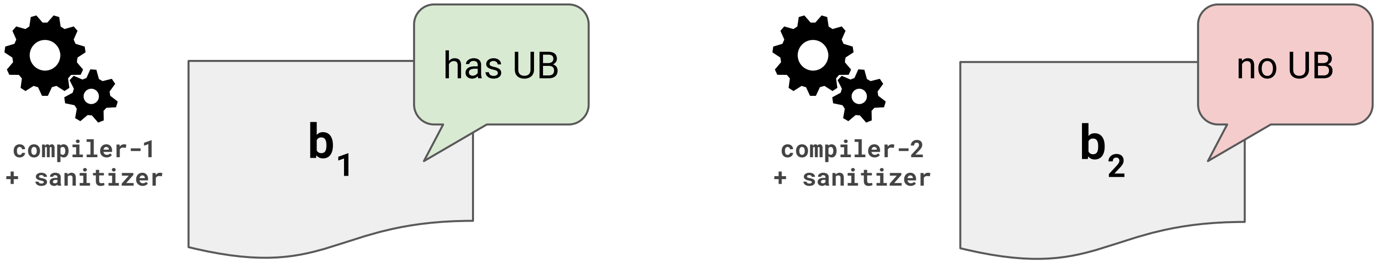
- Check crash/report site of $\mathbf{b_2}$ (potential sanitizer FN bug)

UB Test Oracle with Crash-site Mapping



- Check crash/report site of **b_2** (potential sanitizer FN bug)
- If **b_1** executes the same line
  - **FN bug** in **`compiler-2 + sanitizer`**

UB Test Oracle with Crash-site Mapping



- Check crash/report site of $\mathbf{b_2}$ (potential sanitizer FN bug)
- If $\mathbf{b_1}$ executes the same line
  - **FN bug** in **`compiler-2 + sanitizer`**
- Otherwise
  - Compiler optimization removes UB-inducing code

UBFuzz Key Idea

- **Goal**: Find FN bugs in sanitizer (undetected, but actual UB)
  - **Generate program with UB ✔**
  - Check if sanitizer is able to detect the UB
    - Cannot detect → **sanitizer FN bug**

UBFuzz Key Idea

- <u>**Goal**</u>: Find FN bugs in sanitizer (undetected, but actual UB)
  - **Generate program with UB ✔**
  - Check if sanitizer is able to detect the UB
    - Cannot detect, UB not removed by optimization → **sanitizer FN bug**

UBFuzz Key Idea

- **Goal**: Find FN bugs in sanitizer (undetected, but actual UB)
  - **Generate program with UB ✔**
  - **Check if sanitizer is able to detect the UB ✔**
    - Cannot detect, UB not removed by optimization → **sanitizer FN bug**

Evaluation

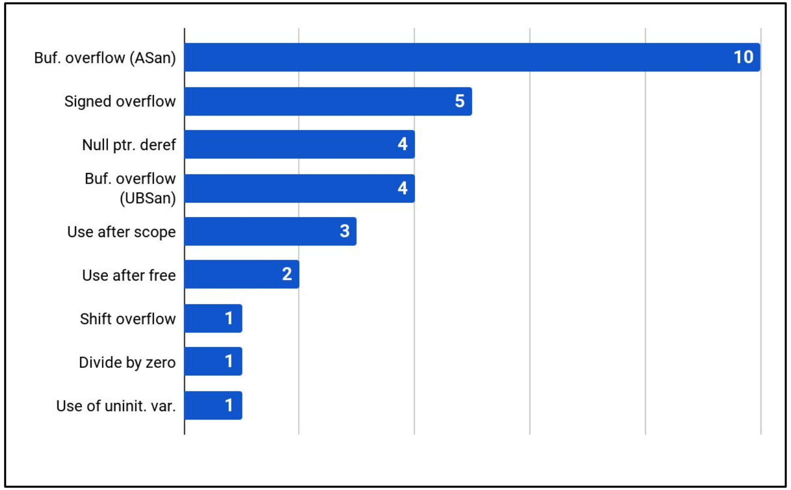
- **Is UBFuzz effective in finding FN bugs in sanitizers?**

Evaluation

- **Is UBFuzz effective in finding FN bugs in sanitizers?**
  - Throughout 5-months testing period, found **31 new bugs**
    - 20 are confirmed, 6 are fixed

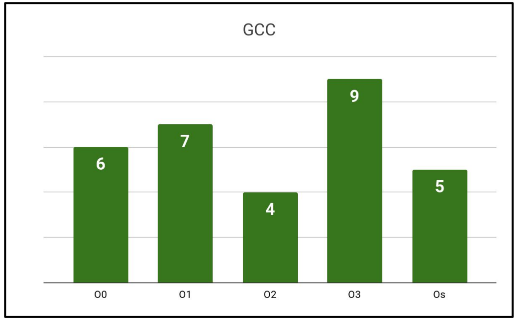
Evaluation

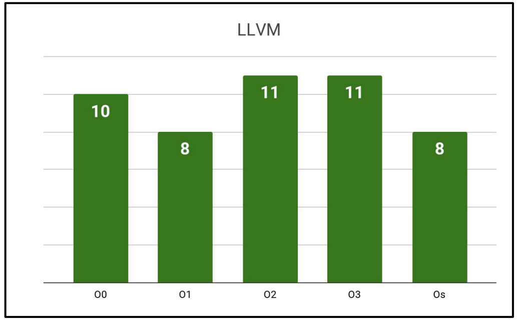
- **Is UBFuzz effective in finding FN bugs in sanitizers?**
  - Throughout 5-months testing period, found **31 new bugs**
    - 20 are confirmed, 6 are fixed
    - Various UB types



Evaluation

- **Is UBFuzz effective in finding FN bugs in sanitizers?**
  - Throughout 5-months testing period, found **31 new bugs**
    - 20 are confirmed, 6 are fixed
    - Various UB types
    - In various optimization levels





Evaluation

- **Is UBFuzz effective in finding FN bugs in sanitizers?**
  - Throughout 5-months testing period, found 31 new bugs
    - 20 are confirmed, 6 are fixed
    - Various UB types
    - In various optimization levels
  - By manual analysis of LLVM-5 and GCC-5's all existing FN bug reports

Evaluation

- **Is UBFuzz effective in finding FN bugs in sanitizers?**
  - Throughout 5-months testing period, found 31 new bugs
    - 20 are confirmed, 6 are fixed
    - Various UB types
    - In various optimization levels
  - By manual analysis of LLVM-5 and GCC-5's all existing FN bug reports
    - UBFuzz was able to find
      - 40% for GCC (16/40)
      - 58% for LLVM (14/24)

Evaluation

- **Is UBFuzz effective in finding FN bugs in sanitizers?**
  - Throughout 5-months testing period, found 31 new bugs
    - 20 are confirmed, 6 are fixed
    - Various UB types
    - In various optimization levels
  - By manual analysis of LLVM-5 and GCC-5's all existing FN bug reports
    - UBFuzz was able to find
      - 40% for GCC (16/40)
      - 58% for LLVM (14/24)

effective

Evaluation

- **How effective is our UB program generator in constructing interesting UB programs?**

Evaluation

- **How effective is our UB program generator in constructing interesting UB programs?**
  - Compare against
    - CSmith + MUSIC (random C code mutator)
    - CSmith-NoSafe (no safe checking in arithmetic logic generation)

Evaluation

- **How effective is our UB program generator in constructing interesting UB programs?**
  - Compare against
    - CSmith + MUSIC (random C code mutator)
    - CSmith-NoSafe (no safe checking in arithmetic logic generation)

Generator	# Gen. Programs w/ UB	# Gen. Programs w/o UB
UBFuzz	**13,872**	**0**
CSmith + MUSIC	704	13,296
CSmith-NoSafe	7,405	6,595

Evaluation

- **How effective is our UB program generator in constructing interesting UB programs?**
  - Compare against
    - CSmith + MUSIC (random C code mutator)
    - CSmith-NoSafe (no safe checking in arithmetic logic generation)

Generator	# Gen. Programs w/ UB	# Gen. Programs w/o UB
UBFuzz	**13,872**	**0**
CSmith + MUSIC	704	13,296
CSmith-NoSafe	7,405	6,595

only arithmetic UBs

Evaluation

- **How effective is our UB program generator in constructing interesting UB programs?**
  - Compare against
    - CSmith + MUSIC (random C code mutator)
    - CSmith-NoSafe (no safe checking in arithmetic logic generation)
  - Juliet Test Suite (collection of UB programs)
    - All 16K programs are detected as UB by sanitizers
    - Not effective to detect sanitizer FN bugs

Conclusion

- **UBFuzz**: novel framework for testing sanitizer implementations
- With **UB program generator** that **inserts shadow statement** from UB free seed programs
- Differential testing is done with **crash-site mapping** as **test oracle**
- UBfuzz has discovered **31 bugs** in ASan, UBSan, and MSan from both GCC and LLVM

Thank you

Steve Gustaman
stevegustaman@kaist.ac.kr

```c
1 int g, *ptr = &g;
2 int **p_ptr = &ptr;
3 int main() {
4   int buf[3]={1,2,3};
5   *ptr = 1;
6   *p_ptr =&buf[3];
7   *ptr = 0xfff;
8 }
```

(a) GCC ASan at -O1 missed the buffer overflow access `*ptr` at line 7. [7]

```c
1 int a, c;
2 short b;
3 long d;
4 int main() {
5     a = (short)(d == c |
6         b > 9) / 0;
7     return a;
8 }
```

(b) GCC's UBSan at all levels missed the division-by-zero at line 5. [9]

```c
1 void b() {
2   int c[1];
3   c;
4 }
5 int main() {
6   int d[1]={1};
7   int *e = d;
8   a = 0;
9   for(;a<=5;++a){
10    int f[1]={};
11    e = f;
12    a||(b(), 1);
13  }
14  return *e;
15 }
```

(c) GCC's ASan missed the use after scope at line 14, where the pointer `e` points to an inner scope variable `f` defined at line 10. [8]

```c
1 volatile int a[5];
2 void b(int x) {
3     if(x)
4       a[5] = 7;
5 }
6 int main(){ b(1); }
```

(d) LLVM's ASan missed the buffer overflow at line 4. [19]

```c
1 int main() {
2     int *a = 0;
3     int b[3]={1, 1, 1};
4     ++b[2];
5     ++(*a);
6 }
```

(e) LLVM's UBSan missed the null pointer dereference at line 5. [20]

```c
1 int main() {
2     unsigned char a;
3     if (a-1)
4         __builtin_printf("boom!\n");
5     return 1;
6 }
```

(f) LLVM's MSan missed the use of uninitialized memory at line 3. [21]

Figure 12. Sample UB programs that trigger sanitizer FN bugs.