

Compiler Validation via Equivalence Modulo Inputs

Vu Le, **Mehrdad Afshari**, Zhendong Su

University of California, Davis

Compiler bugs are evil

Compiler bugs are scary

Compiler bugs are hard to catch

What can we do? What can we do?

EMI is our answer

Orion comes to rescue

200+ bugs found within a year

llvm bug 14972

```
struct tiny { char c; char d; char e; };

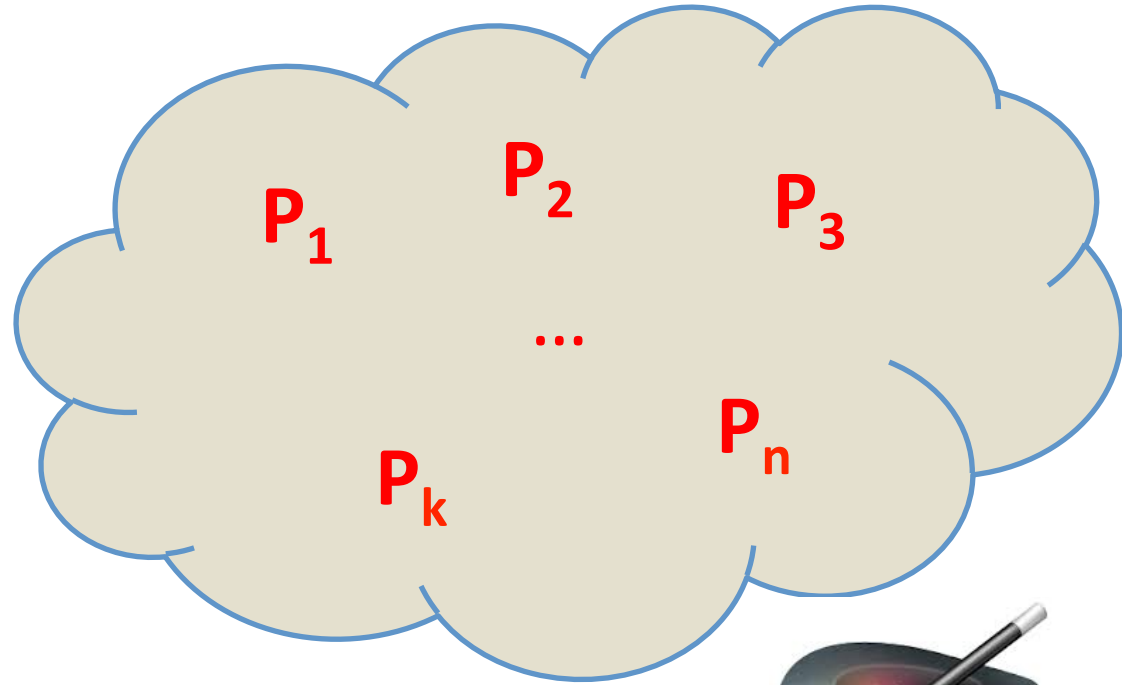
void foo(struct tiny x) {
    if (x.c != 1) abort();
    if (x.e != 1) abort();
}

int main() {
    struct tiny s;
    s.c = 1; s.d = 1; s.e = 1;
    foo(s);
    return 0;
}
```

```
$ clang -m32 -O0 test.c ; ./a.out
$ clang -m32 -O1 test.c ; ./a.out
Aborted (core dumped)
```

vision

P \equiv



key challenges

□ Generation

- ◆ How to generate **different** but **equivalent** tests?

□ Validation

- ◆ How to check that tests are **indeed equivalent**?

□ Both are long-standing hard issues

equiv. modulo inputs



□ Relax equiv. wrt a **given input**

◆ Variants must satisfy $P(i) = P_k(i)$ on input i

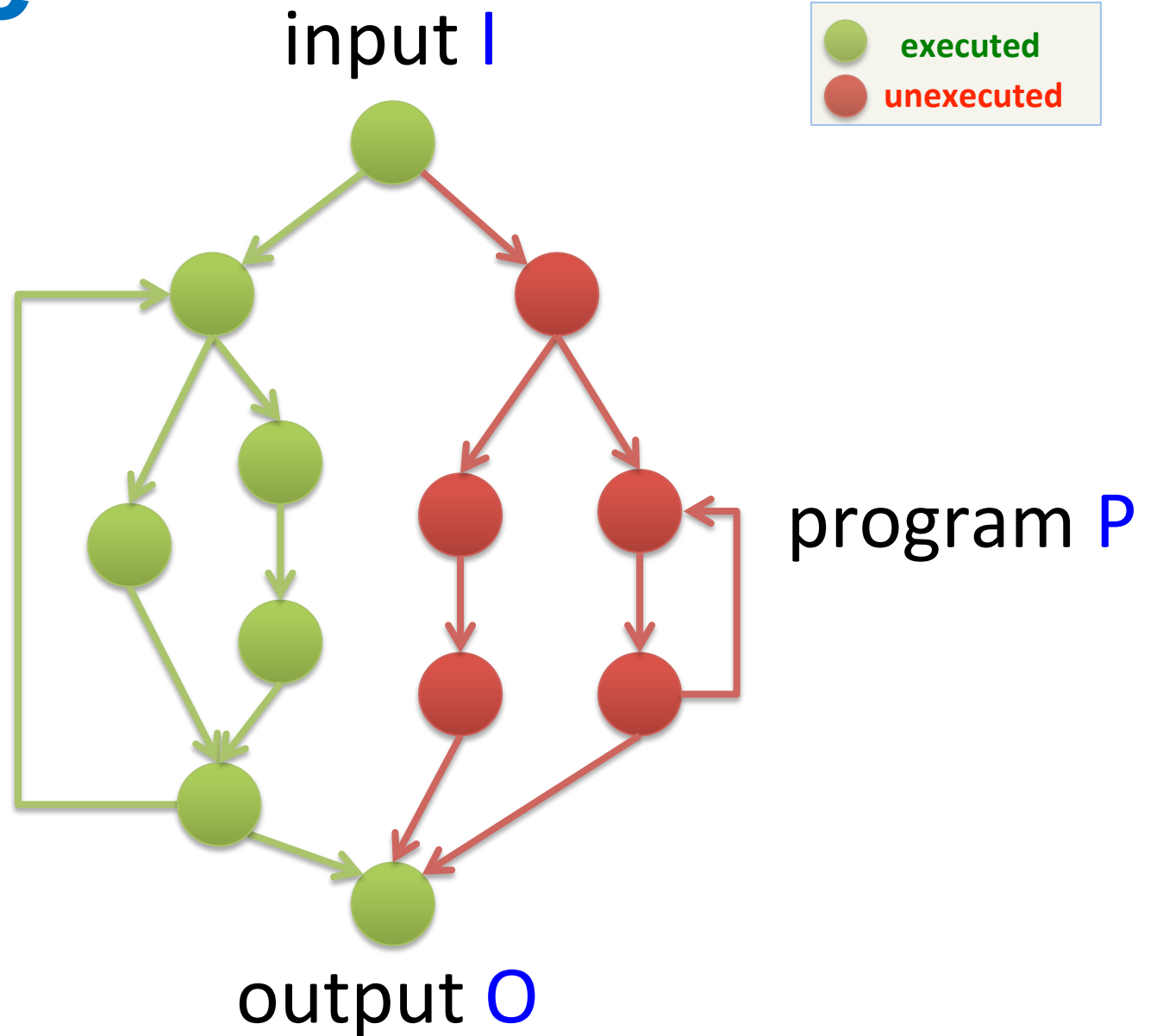
◆ But may differ on other input j : $P(j) \neq P_k(j)$

□ Exploit close **interplay** between

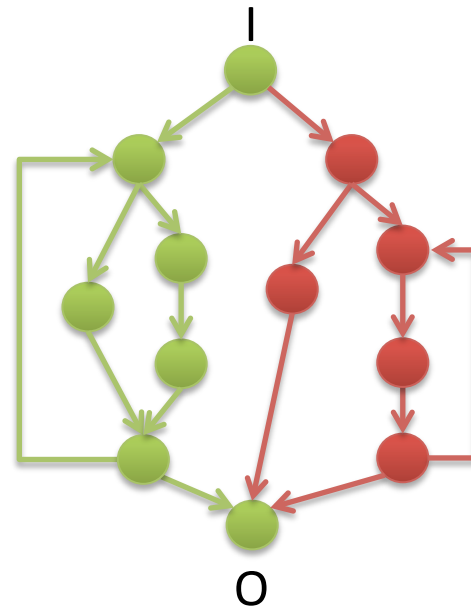
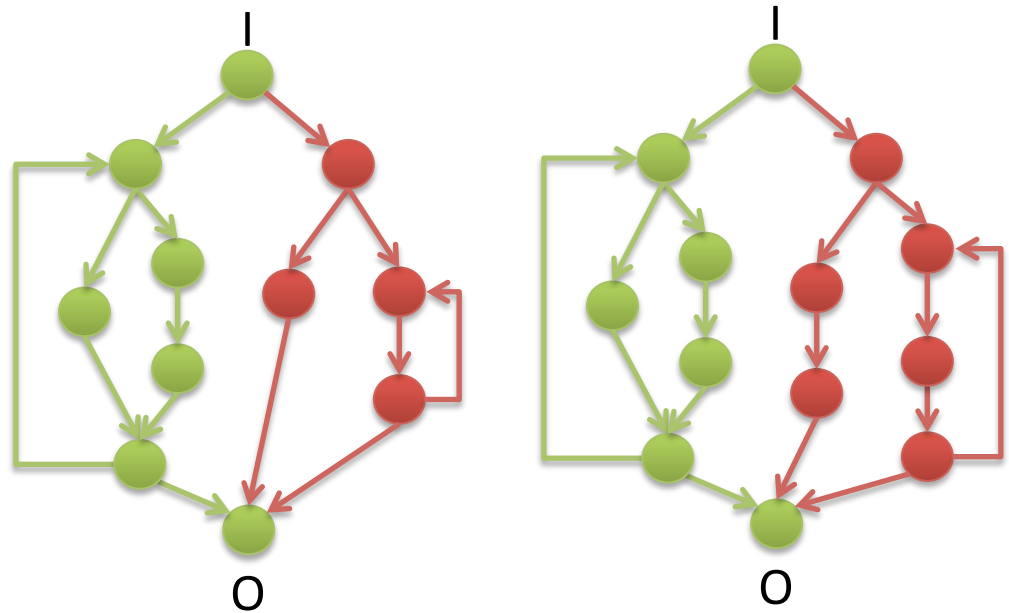
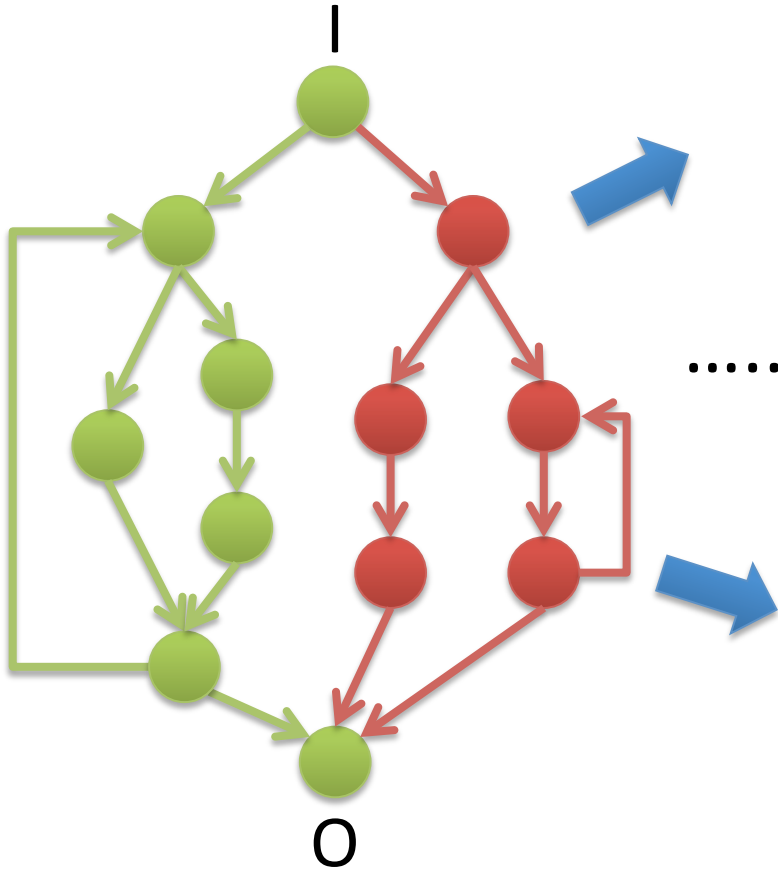
◆ **Dynamic** program execution on **some input**

◆ **Static** compilation for **all input**

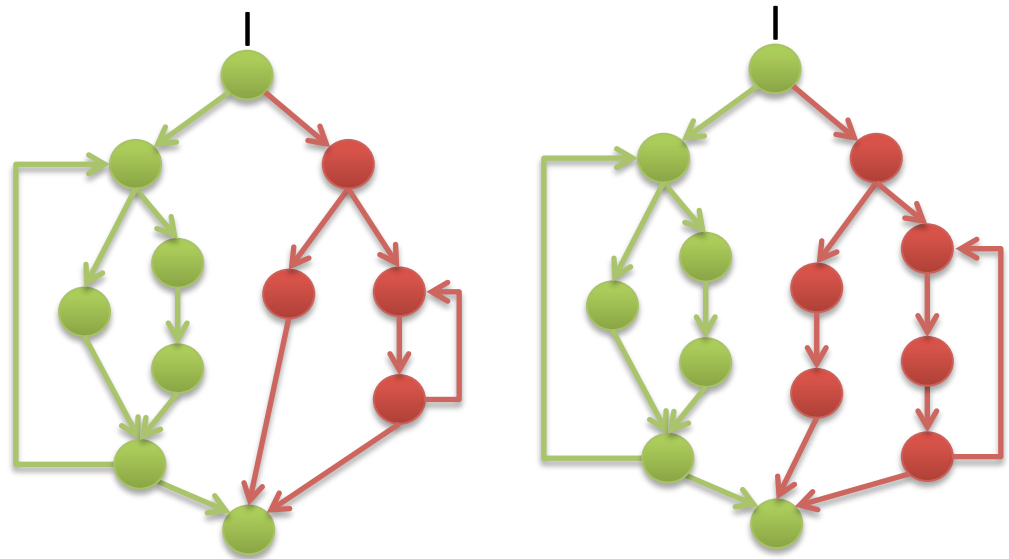
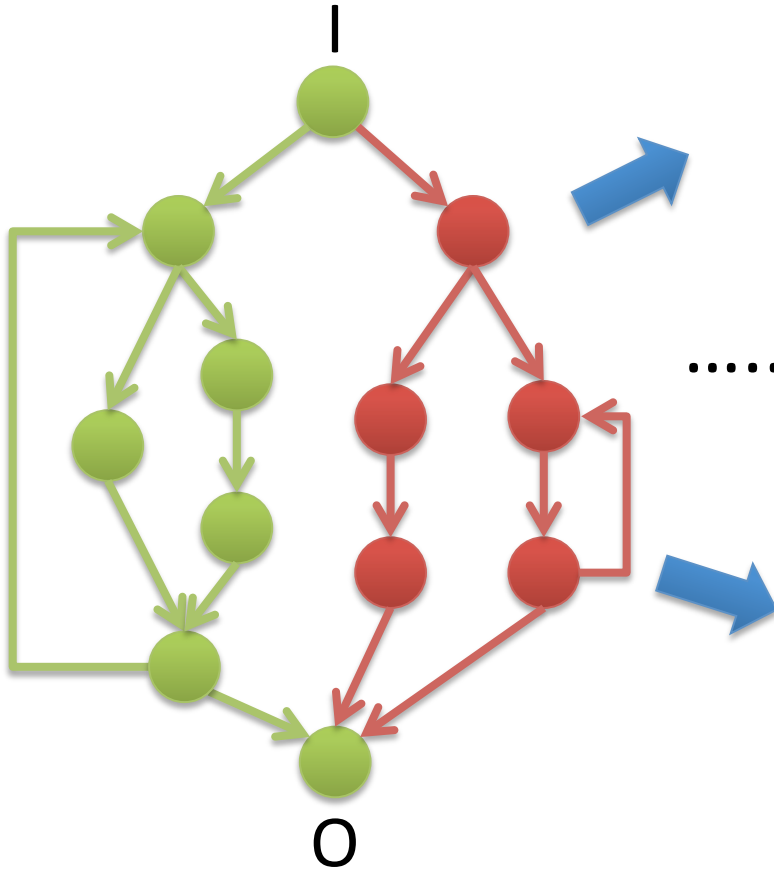
profile



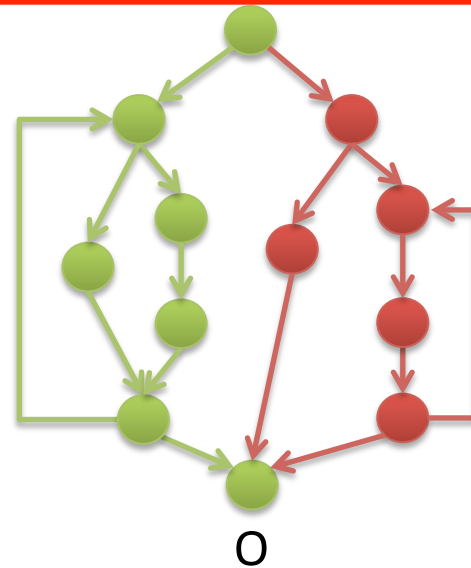
mutate



mutate



equivalent wrt 1



revisit challenges

□ Generation (**easy**)

- ◆ How to generate **different** but **equivalent** tests?

□ Validation (**easy**)

- ◆ How to check that tests are **indeed equivalent**?

□ ~~Both are long-standing hard issues~~

llvm bug 14972

```
struct tiny { char c; char d; char e; };

void foo(struct tiny x) {
    if (x.c != 1) abort();
    if (x.e != 1) abort();
}

int main() {
    struct tiny s;
    s.c = 1; s.d = 1; s.e = 1;
    foo(s);
    return 0;
}
```

```
$ clang -m32 -O0 test.c ; ./a.out
$ clang -m32 -O1 test.c ; ./a.out
Aborted (core dumped)
```

seed file

```
struct tiny { char c; char d; char e; };
f(int n, struct tiny x, struct tiny y,
  struct tiny z, long l) {
    if (x.c != 10) abort();
    if (x.d != 20) abort();
    if (x.e != 30) abort();
    if (y.c != 11) abort();
    if (y.d != 21) abort();
    if (y.e != 31) abort();
    if (z.c != 12) abort();
    if (z.d != 22) abort();
    if (z.e != 32) abort();
    if (l != 123) abort();
}
main() {
    struct tiny x[3];
    x[0].c = 10;
    x[1].c = 11;
    x[2].c = 12;
    x[0].d = 20;
    x[1].d = 21;
    x[2].d = 22;
    x[0].e = 30;
    x[1].e = 31;
    x[2].e = 32;
    f(3, x[0], x[1], x[2], (long)123);
    exit(0);
}
```

```
$ clang -m32 -O0 test.c ; ./a.out
$ clang -m32 -O1 test.c ; ./a.out
```

seed file

```
struct tiny { char c; char d; char e; };
f(int n, struct tiny x, struct tiny y,
  struct tiny z, long l) {
    if (x.c != 10) abort();
    if (x.d != 20) abort();
    if (x.e != 30) abort();
    if (y.c != 11) abort();
    if (y.d != 21) abort();
    if (y.e != 31) abort();
    if (z.c != 12) abort();
    if (z.d != 22) abort();
    if (z.e != 32) abort();
    if (l != 123) abort();
}
main() {
    struct tiny x[3];
    x[0].c = 10;
    x[1].c = 11;
    x[2].c = 12;
    x[0].d = 20;
    x[1].d = 21;
    x[2].d = 22;
    x[0].e = 30;
    x[1].e = 31;
    x[2].e = 32;
    f(3, x[0], x[1], x[2], (long)123);
    exit(0);
}
```

← unexecuted

```
$ clang -m32 -O0 test.c ; ./a.out
$ clang -m32 -O1 test.c ; ./a.out
```

transformed file

```
struct tiny { char c; char d; char e; };
f(int n, struct tiny x, struct tiny y,
  struct tiny z, long l) {
    if (x.c != 10) /* deleted */;
    if (x.d != 20) abort();
    if (x.e != 30) /* deleted */;
    if (y.c != 11) abort();
    if (y.d != 21) abort();
    if (y.e != 31) /* deleted */;
    if (z.c != 12) abort();
    if (z.d != 22) /* deleted */;
    if (z.e != 32) abort();
    if (l != 123) /* deleted */;
}
main() {
    struct tiny x[3];
    x[0].c = 10;
    x[1].c = 11;
    x[2].c = 12;
    x[0].d = 20;
    x[1].d = 21;
    x[2].d = 22;
    x[0].e = 30;
    x[1].e = 31;
    x[2].e = 32;
    f(3, x[0], x[1], x[2], (long)123);
    exit(0);
}
```

```
$ clang -m32 -O0 test.c ; ./a.out
$ clang -m32 -O1 test.c ; ./a.out
Aborted (core dumped)
```

reduced file

```
struct tiny { char c; char d; char e; };

void foo(struct tiny x) {
    if (x.c != 1) abort();
    if (x.e != 1) abort();
}

int main() {
    struct tiny s;
    s.c = 1; s.d = 1; s.e = 1;
    foo(s);
    return 0;
}
```


```
$ clang -m32 -O0 test.c ; ./a.out
$ clang -m32 -O1 test.c ; ./a.out
Aborted (core dumped)
```

llvm bug autopsy

```
struct tiny { char c; char d; char e; };
```

```
void foo(struct tiny x) {  
    if (x.c != 1) abort();  
    if (x.e != 1) abort();  
}
```

GVN: load struct
using 32-bit load



```
int main() {  
    struct tiny s;  
    s.c = 1; s.d = 1; s.e = 1;  
    foo(s);  
    return 0;  
}
```

```
$ clang -m32 -O0 test.c ; ./a.out  
$ clang -m32 -O1 test.c ; ./a.out  
Aborted (core dumped)
```



llvm bug autopsy

```
struct tiny { char c; char d; char e; };
```


```
void foo(struct tiny x) {  
    if (x.c != 1) abort();  
    if (x.e != 1) abort();  
}
```

```
int main() {  
    struct tiny s;  
    s.c = 1; s.d = 1; s.e = 1;  
    foo(s);  
    return 0;  
}
```

GVN: load struct
using 32-bit load



SRoA: read past
the struct's end



→
undefined
behavior


```
$ clang -m32 -O0 test.c ; ./a.out  
$ clang -m32 -O1 test.c ; ./a.out  
Aborted (core dumped)
```

llvm bug autopsy

```
struct tiny { char c; char d; char e; };
```


```
void foo(struct tiny x) {  
    if (x.c != 1) abort();  
    if (x.e != 1) abort();  
}
```

GVN: load struct
using 32-bit load




```
int main() {  
    struct tiny s;  
    s.c = 1; s.d = 1; s.e = 1;  
    foo(s);  
    return 0;  
}
```

SRoA: read past
the struct's end



→
undefined
behavior



remove



```
$ clang -m32 -O0 test.c ; ./a.out  
$ clang -m32 -O1 test.c ; ./a.out  
Aborted (core dumped)
```

developers

*“... very, very concerning when I
got to the root cause, and very
annoying to fix ...”*

http://llvm.org/bugs/show_bug.cgi?id=14972

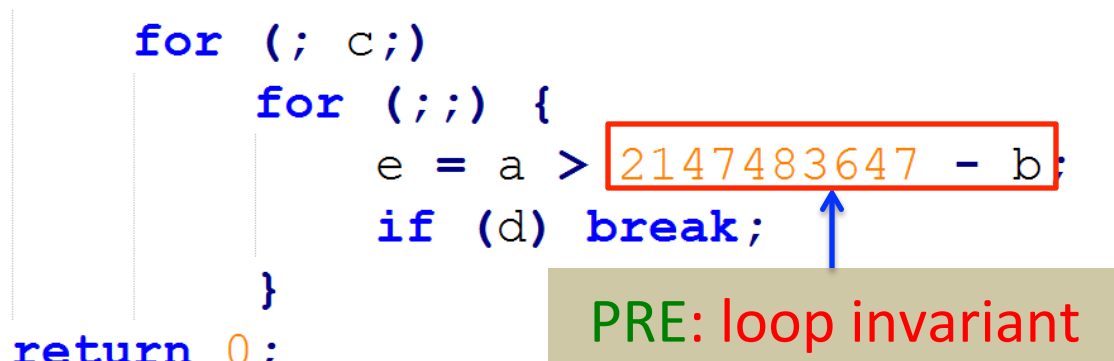
gcc bug 58731

```
int a, b, c, d, e;
int main() {
    for (b = 4; b > -30; b--)
        for (; c;)
            for (;;) {
                e = a > 2147483647 - b;
                if (d) break;
            }
    return 0;
}
```

```
$ gcc -O0 test.c ; ./a.out
$ gcc -O3 test.c ; ./a.out
^C
```

gcc bug autopsy

```
int a, b, c, d, e;
int main() {
    for (b = 4; b > -30; b--)
        for (; c;)
            for (;;) {
                e = a > 2147483647 - b;
                if (d) break;
            }
    return 0;
}
```



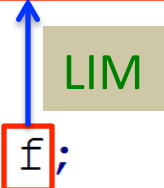
A diagram illustrating the nested loops in the code. Three vertical lines connect the opening and closing braces of the three nested loops: the outermost loop (for b), the middle loop (for c), and the innermost loop (for ;). A blue arrow points from a text box below to the expression `2147483647 - b` in the assignment statement `e = a > 2147483647 - b;`.

PRE: loop invariant

```
$ gcc -O0 test.c ; ./a.out
$ gcc -O3 test.c ; ./a.out
^C
```

gcc bug autopsy

```
int a, b, c, d, e;
int main() {
    for (b = 4; b > -30; b--)
        int f = 2147483647 - b;
        for (; c;)
            for (;;) {
                e = a > f;
                if (d) break;
            }
    return 0;
}
```



A diagram illustrating a bug in GCC. A green box labeled "LIM" has a blue arrow pointing upwards to the variable `f` in the expression `e = a > f;`. The variable `f` is also enclosed in a red box. The value `2147483647` in the line `int f = 2147483647 - b;` is also enclosed in a red box.

```
$ gcc -O0 test.c ; ./a.out
$ gcc -O3 test.c ; ./a.out
^C
```

gcc bug autopsy

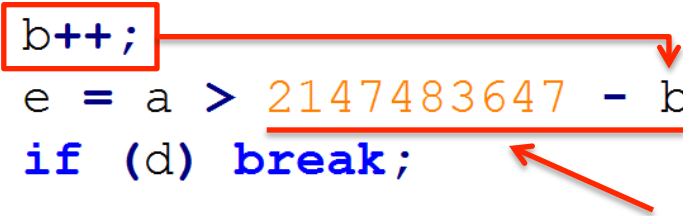
```
int a, b, c, d, e;
int main() {
    for (b = 4; b > -30; b--)
        int f = 2147483647 - b;
        for (; c;)
            for (;;) {
                e = a > f;
                if (d) break;
            }
    return 0;
}
```

integer overflow

```
$ gcc -O0 test.c ; ./a.out
$ gcc -O3 test.c ; ./a.out
^C
```

seed program

```
int a, b, c, d, e;
int main() {
    for (b = 4; b > -30; b--)
        for (; c;)
            for (;;) {
                b++;
                e = a > 2147483647 - b;
                if (d) break;
            }
    return 0;
}
```



no longer a loop invariant

```
$ gcc -O0 test.c ; ./a.out
$ gcc -O3 test.c ; ./a.out
```


why effective?

- Compilers produce correct code for **all input**

why effective?

- Compilers produce correct code for all input
- Variants have different data & control flow
 - ◆ Exercise **various** optimization strategies
 - ◆ Demand exact **same output** on the given input

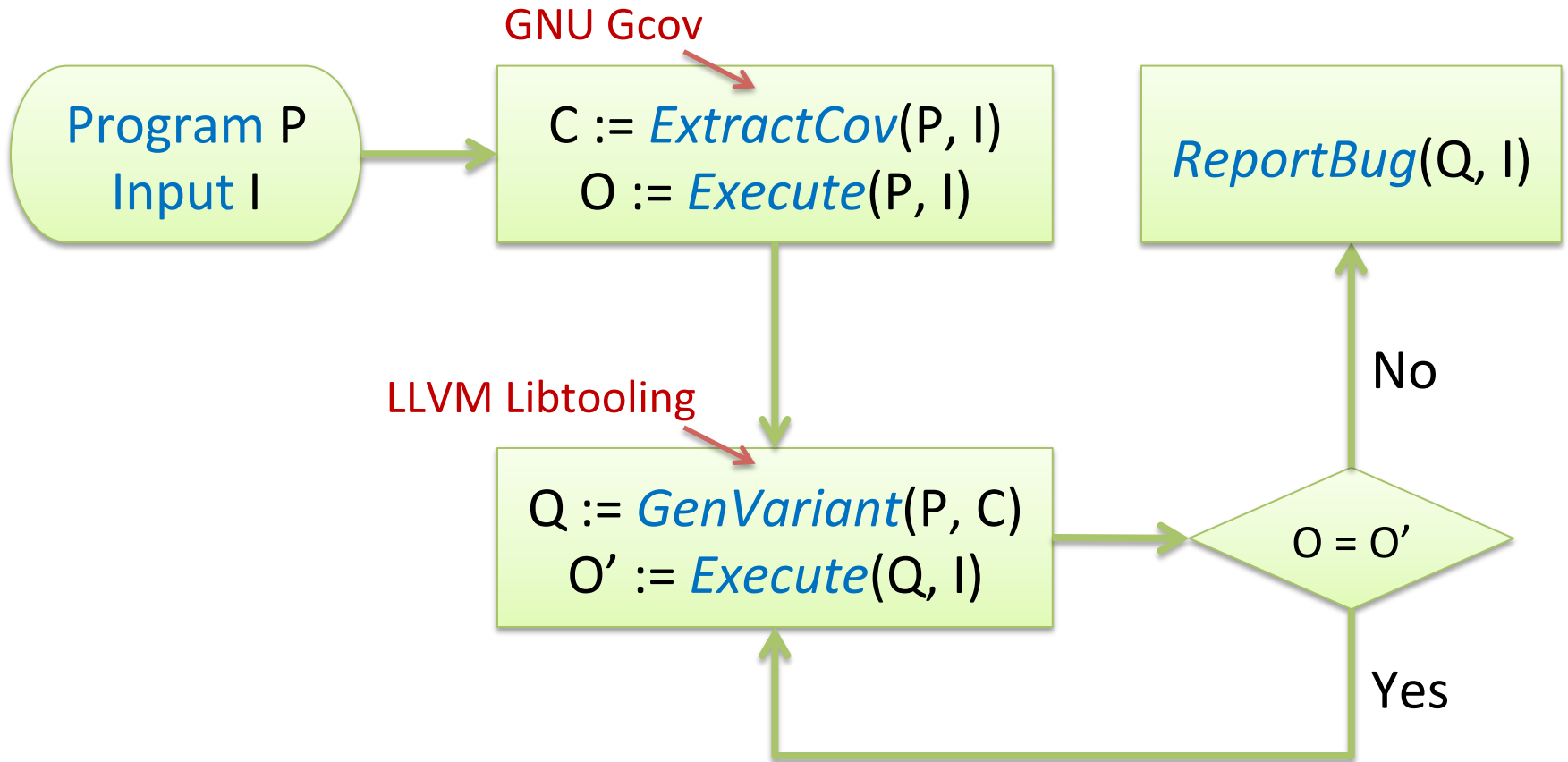
orion

- A practical realization of EMI

orion

- ❑ A practical realization of EMI
- ❑ Targeting C compilers
 - ◆ Randomly **prune** unexecuted code
 - ◆ Extremely **effective**

orion



evaluation

- ❑ Two multi-core Ubuntu machines
- ❑ April 2013 – March 2014
- ❑ Seed programs
 - ◆ Compiler regression test suites
 - ◆ Open-source projects
 - ◆ Csmith-generated programs

bug counts

	GCC	LLVM	TOTAL
Reported	111	84	195
Marked Duplicate	28	7	35
Confirmed	79	68	147
Fixed	56	54	110

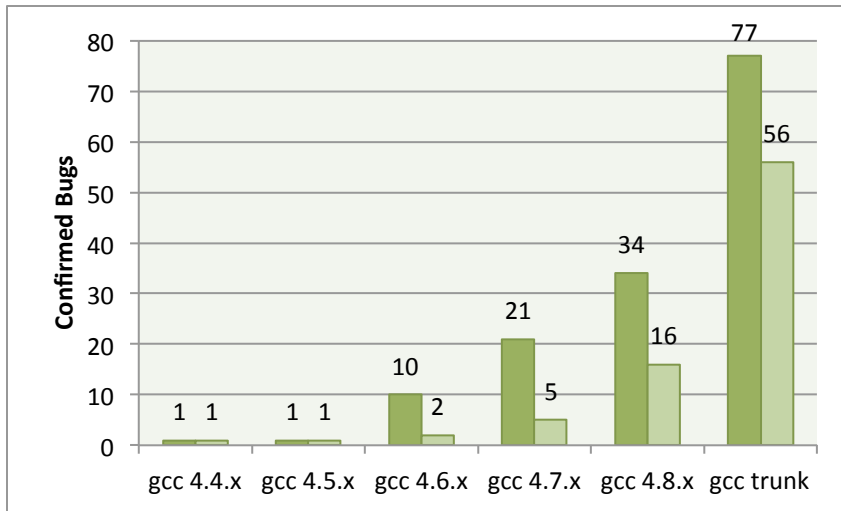
bug types

	GCC	LLVM	TOTAL
Wrong code	46	49	95
Crash	23	10	33
Performance	10	9	19

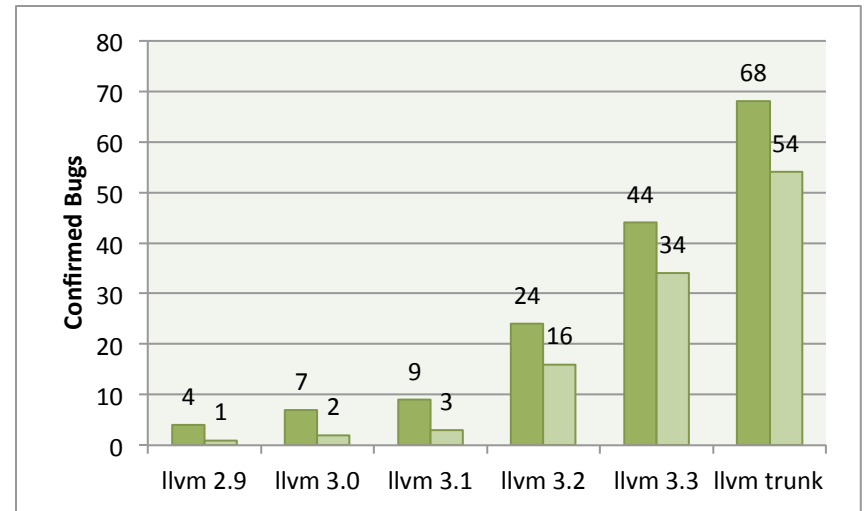
bug importance

- ❑ Most bugs have already been **fixed**
- ❑ Many were **critical, release-blocking**
- ❑ Some affected **real-world** projects

affected versions

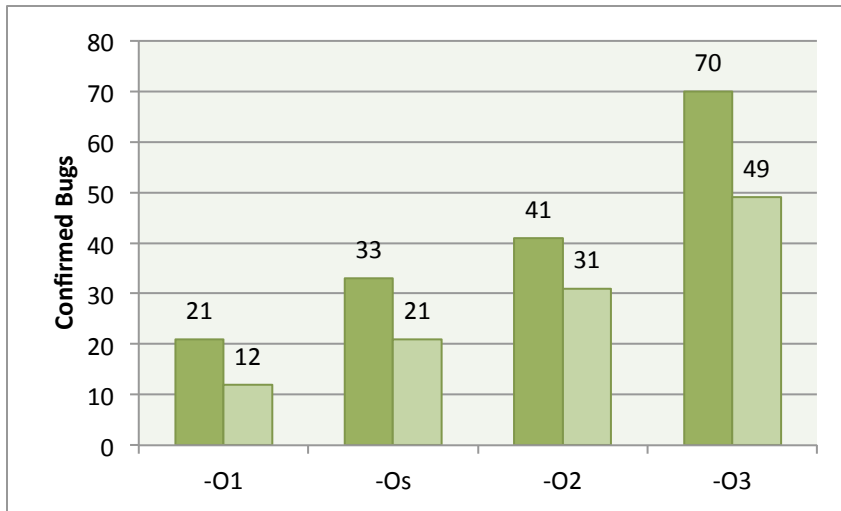


GCC

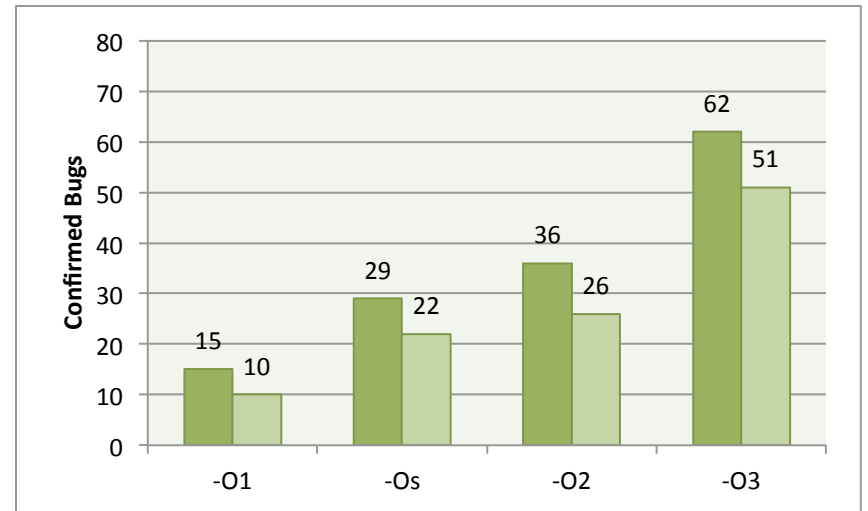


LLVM

affected opt. levels

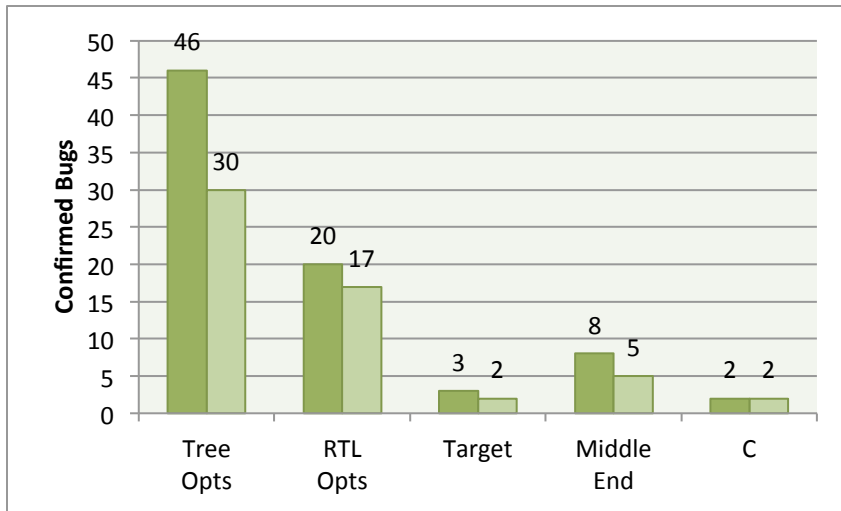


GCC

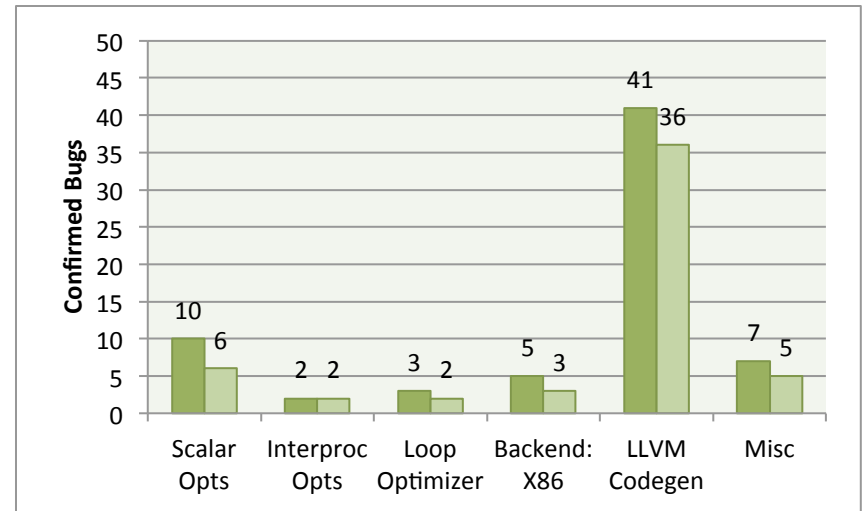


LLVM

affected components



GCC



LLVM

related work

- ❑ Verified compiler
- ❑ Translation validation
- ❑ Random program generation

future work

- ❑ Investigate other **mutation strategies**
- ❑ Extend EMI to handle **floating-point** code
- ❑ Adapt EMI to other **languages & settings**

conclusion

- ❑ EMI is **general** and **widely applicable**
 - ◆ Can test compilers, analysis and transformation tools
 - ◆ Generates real-world tests
 - ◆ Requires no reference compilers
- ❑ Orion is very **effective**
 - ◆ Has uncovered **200+ bugs** in GCC and LLVM
 - ◆ Majority of the bugs were **miscompilations**

conclusion

- ❑ EMI is **general** and **widely applicable**
 - ◆ Can test compilers, analysis and transformation tools
 - ◆ Generates real-world tests
 - ◆ Requires no reference compilers
- ❑ Orion is very **effective**
 - ◆ Has uncovered **200+ bugs** in GCC and LLVM
 - ◆ Majority of the bugs were **miscompilations**

Exciting new direction with **many applications**