Addressing the Saturation Effect in Compiler Testing

Cristian Cadar

SOFTWARE RELIABILITY GROUP

Department of Computing
Imperial College London

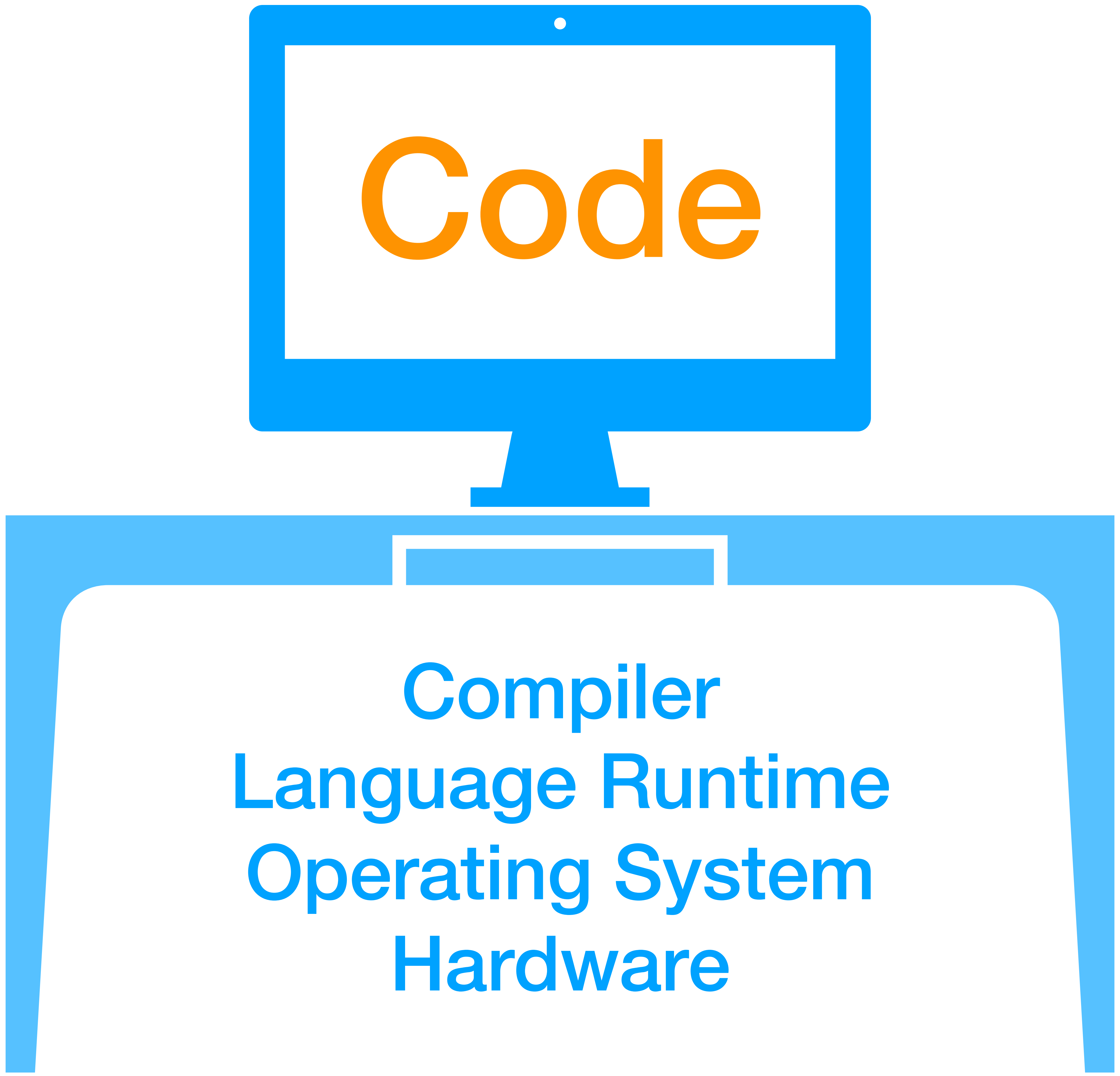
Based on joint work with Karine Even-Mendoza, Arindam Sharma and Alastair Donaldson

Funded by Engineering and Physical Sciences Research Council, erc European Research Council

National University of Singapore
12 May 2023

Trusted Development Base

(Typically Hidden from View)



Compilers are special

“I know it’s me!”

😅

Compilers are special

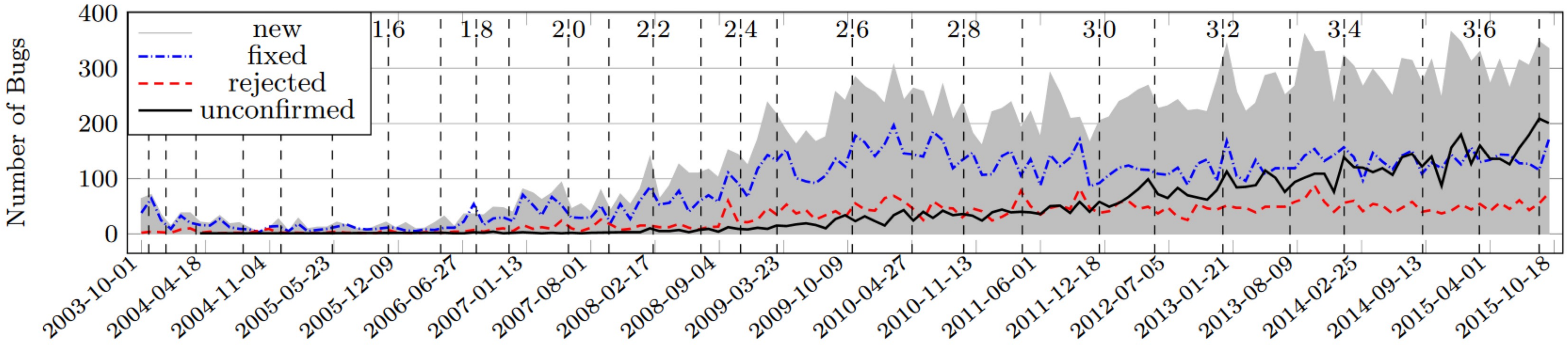
```
address = optimizer.optimizeExpr(address, true);
StatePair zeroPointer = fork(state, Expr::createIsZero(address), true);
if (zeroPointer.first) {
  if (target)
    bindLocal(target, *zeroPointer.first, Expr::createPointer(0));
}
if (zeroPointer.second) { // address != 0
  ExactResolutionList rl;
  resolveExact(*zeroPointer.second, address, rl, "free");

  for (Executor::ExactResolutionList::iterator it = rl.begin(),
         ie = rl.end(); it != ie; ++it) {
    const MemoryObject *mo = it->first.first;
    if (mo->isLocal) {
      terminateStateOnError(*it->second, "free of alloca", Free, NULL,
                            getAddressInfo(*it->second, address));
    } else if (mo->isGlobal) {
      terminateStateOnError(*it->second, "free of global", Free, NULL,
                            getAddressInfo(*it->second, address));
    } else {
      it->second->addressSpace.unbindObject(mo);
      if (target)
        bindLocal(target, *it->second, Expr::createPointer(0));
```

"I proved this source code correct!"

😅

Compilers Are Also Just Software

- Complex software: both GCC and LLVM/Clang have millions of LOCs
- Over 100 bugs/months fixed on average in both compilers recently



History of LLVM Bug Tracking System (2003-2015). Taken from *[Sun et al., ISSTA'16]*

Not All Bugs are the Same

gcc.gnu.org/bugzilla/show_bug.cgi?id=98630

```
When compiling with GCC-10 (gcc-10 (Ubuntu 10.2.0-5ubuntu1~20.04) 10.2.0):
> gcc-10 -w -O2 r.c -pedantic -Wall -Wextra
> ./a.out
> Segmentation fault (core dumped)
```

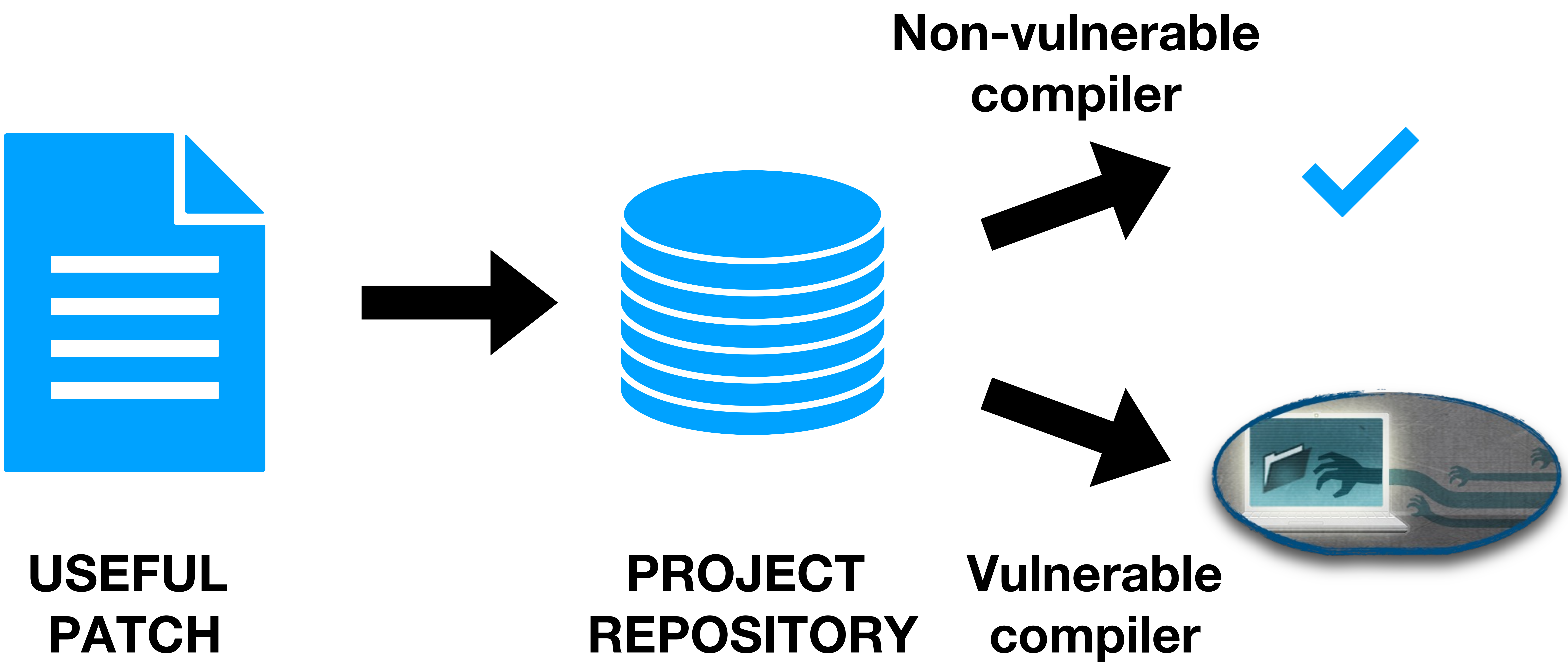
CRASH BUG

gcc.gnu.org/bugzilla/show_bug.cgi?id=94809

```
Seen on: 18.04.4 LTS
kar@kar-VirtualBox:~/ex1$ gcc-9 ex2.c -o ex
kar@kar-VirtualBox:~/ex1$ ./ex
0
kar@kar-VirtualBox:~/ex1$ gcc-6 ex2.c -o ex
kar@kar-VirtualBox:~/ex1$ ./ex
1
```

MISCOMPILATION

The Security Angle: Trusting Trust

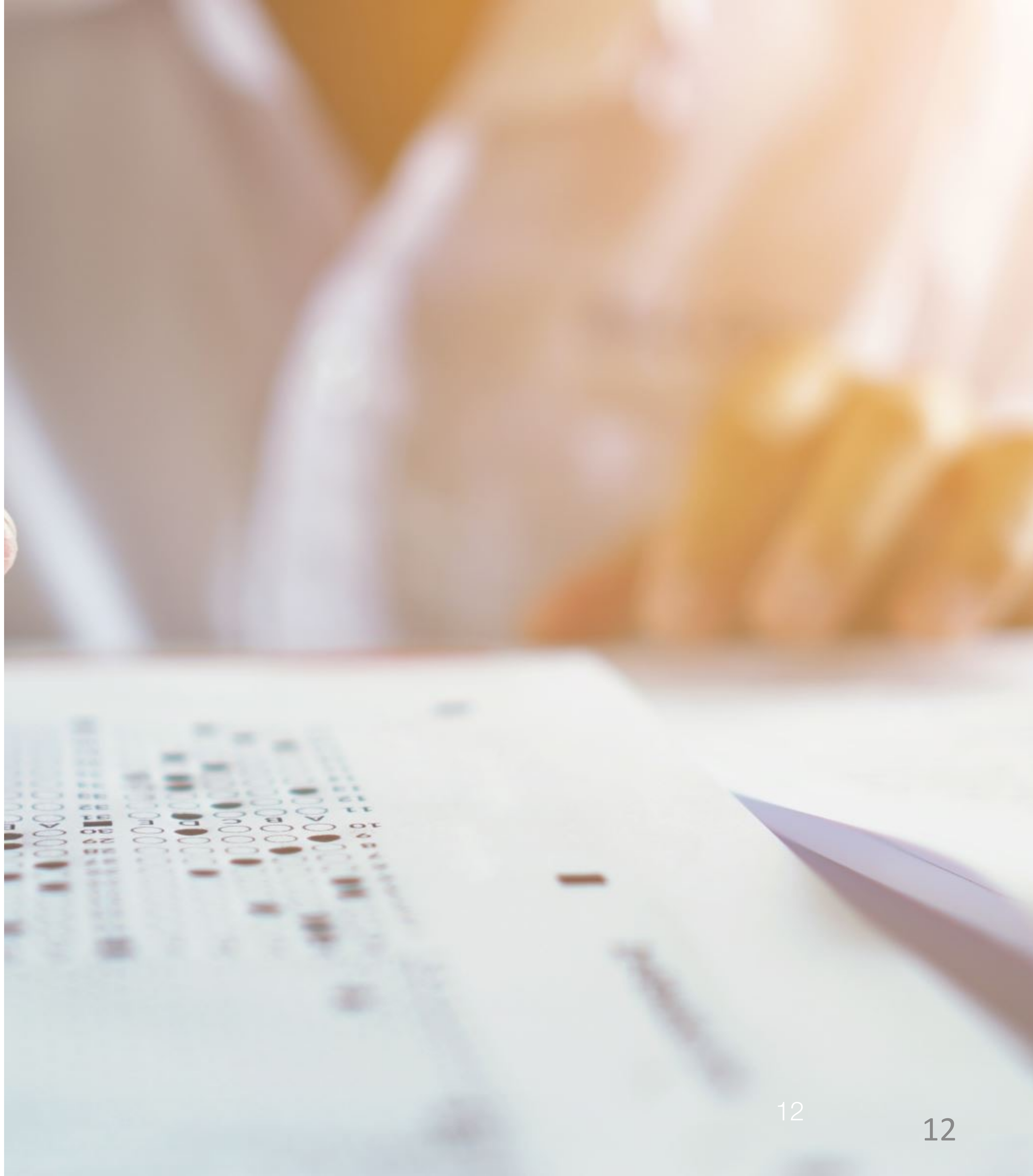


What Makes Compiler* Testing Hard?

At least the following related factors:

- Absence of an oracle
- Undefined, unspecified and implementation-defined behaviour
- Nondeterminism
- Lack of clear language semantics

*** We use the term compiler in a broad sense, including other language processors such as program analysis tools**



12

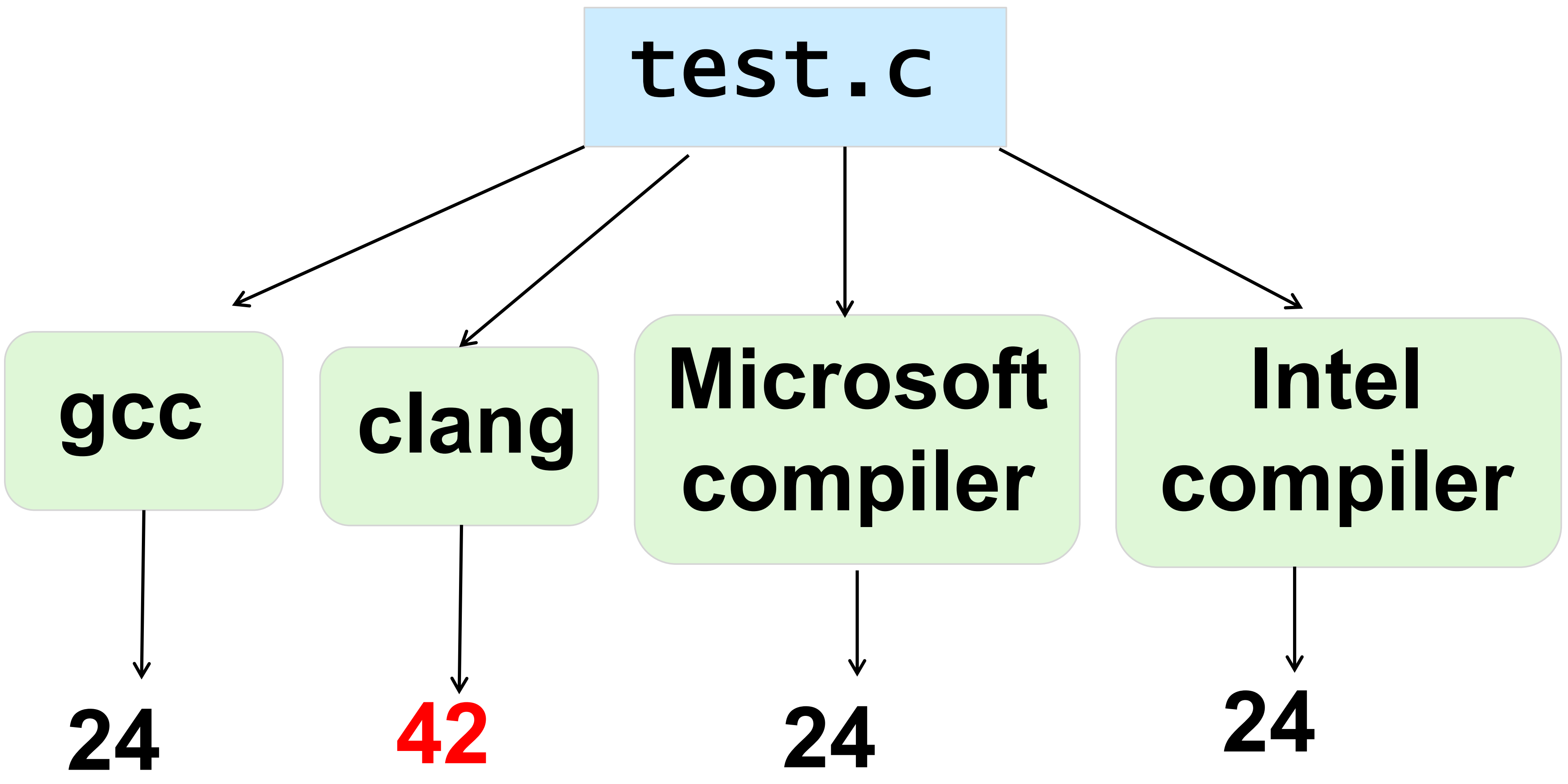
Derived Oracle: Differential Testing

Compile a program with many compilers

Compare results

Mismatches indicate bugs

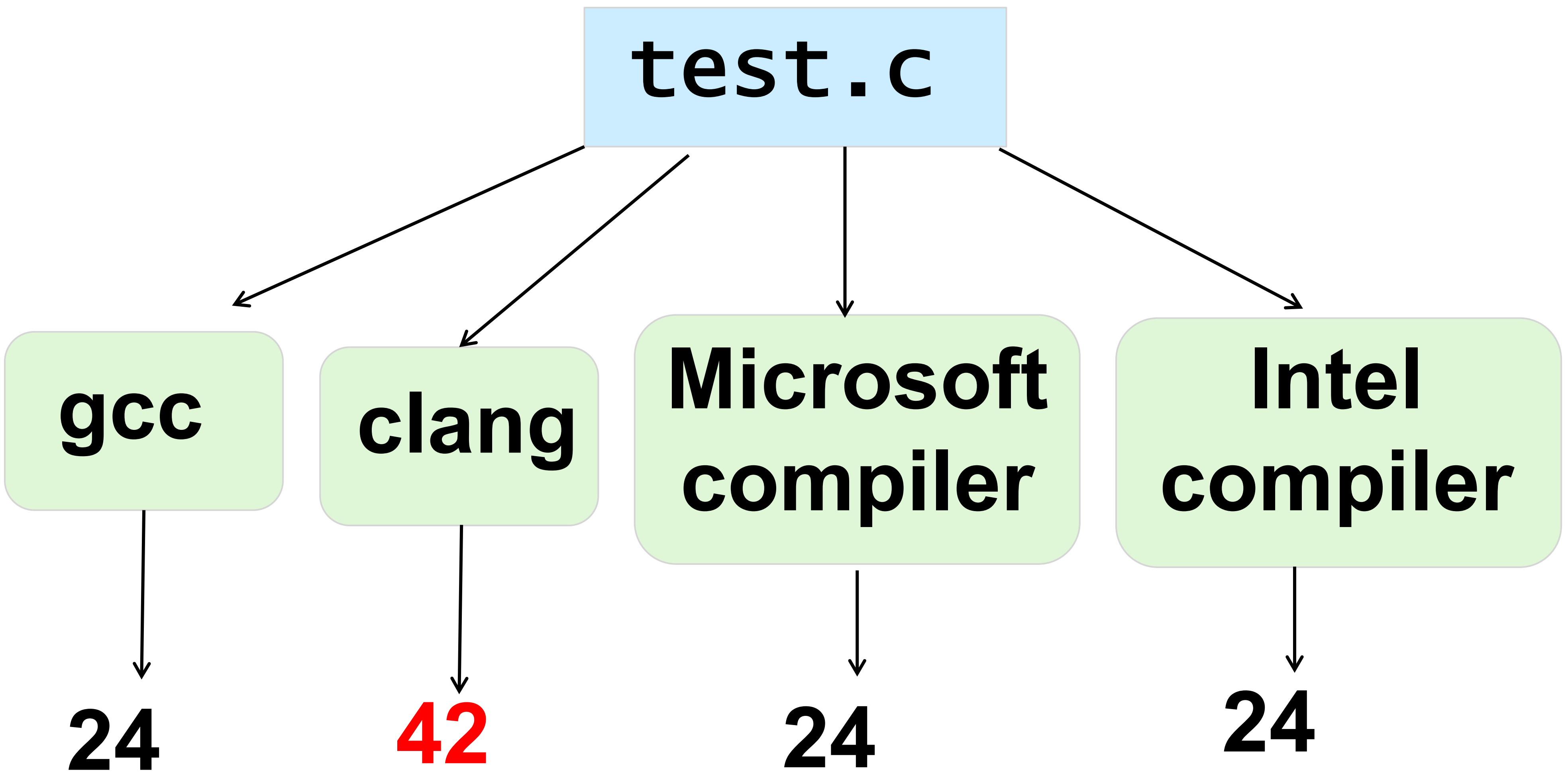
Majority is probably right



Differential Testing Requirements

Program must be:
- **Deterministic**
- **Free from undefined / unspecified behaviour**

Compilers:
- **Must agree on implementation-defined behaviour**



Crosschecking multiple compiler optimization levels also works well; e.g. `gcc -O0` **vs.** `gcc -O2`

Undefined Behaviour

- **Undefined behaviour:** the standard imposes no requirement, i.e. the compiler can generate any code (or no code at all)
    - E.g., buffer overflows, uninitialized reads, signed integer overflow

Example: Saturating Add

Let’s try to implement saturating addition for signed integers in C

x + y is clamped to an extreme value if it falls outside the signed integer range

```c
int saturating_add(int x, int y) {
  if (x > 0 && y > 0 && x + y < 0)
    return INT_MAX;
  if (x < 0 && y < 0 && x + y > 0)
    return INT_MIN;
  return x + y;
}
```

Saturating Add in Action

Compiled with gcc 7.5.0, -O0, we get:

Intended saturating behaviour (32-bit)

```
saturating_add(1, 2) == 3
saturating_add(-5, 2) == -3
saturating_add(1000000000, 1000000000) == 2000000000
saturating_add(2000000000, 2000000000) == 2147483647
saturating_add(-2000000000, -2000000000) == -2147483648
```

Compiled with gcc 7.5.0, -O2, we get:

```
saturating_add(1, 2) == 3
saturating_add(-5, 2) == -3
saturating_add(1000000000, 1000000000) == 2000000000
saturating_add(2000000000, 2000000000) == -294967296
saturating_add(-2000000000, -2000000000) == 294967296
```

This looks like wrap-around behaviour!

The Compiler’s “Thought Process”

“I will assume this program does not exhibit undefined behaviours, because if it does then it matters not what code I emit.”

```
int saturating_add(int x, int y) {
  if (x > 0 && y > 0 && x + y < 0)
    return INT_MAX;
  if (x < 0 && y < 0 && x + y > 0)
    return INT_MIN;
  return x + y;
}
```

“I know `x + y` does not overflow: this would be an UB.

So if `x` and `y` are positive, `x + y` must be positive.

The condition is equivalent to *false*.

***Excellent!!*”**

“By similar reasoning, this condition is equivalent to *false.*”

The Compiler's "Thought Process"

"I can simplify the program to:"

```
int saturating_add(int x, int y) {
  if (false)
    return INT_MAX;
  if (false)
    return INT_MIN;
  return x + y;
}
```

"Or better still, to:"

```
int saturating_add(int x, int y) {
  return x + y;
}
```

Full marks, compiler

C Compiler Fuzzing

Blackbox Fuzzing	Greybox Fuzzing	Whitebox Fuzzing
Highly successful, but starting to saturate	**Limited success, finding mostly front-end bugs**	**None available**

Blackbox C Compiler Testing

Hundreds of crash and miscompilation bugs found in GCC, LLVM by several effective blackbox fuzzers

Csmith

EMI Orange

intel / yarpgen

Csmith

Csmith has found hundreds of bugs in GCC and LLVM

Csmith team won Most Influential PLDI 2011 Paper Award (at PLDI 2021)

Compilers Have Become Immune to Csmith

John Regehr (Csmith research group lead) in 2019:

> **John Regehr**
> @johnregehr
>
> I hadn't run Csmith for a while and it turns out LLVM is now amazingly resistant to it, ran a million tests overnight without finding a crash or miscompile
>
> 5:59 pm · 1 Jun 2019 · Twitter Web App
>
> **6** Retweets **64** Likes

Similar story for other compiler fuzzing tools

Csmith and Undefined Behaviour

- Csmith introduces constraints for UB-free program generation

- Example: avoid UB related to division in zero via "safe math" wrappers

$$a/b \longrightarrow \underbrace{(b == 0)\,?\,a : a/b}$$

Unsafe division **Safe division wrapper**

```c
int main()
{
  int s = 5;
  int t = 2147483646;
  for (int i = 8; i >= -8; i--) {
    s = s+i;
    t = t/i;
  }
  printf("Result: %d,%d\n", s,t);
}
```

```c
int main()
{
  int s = 5;
  int t = 2147483646;
  for (int i = 8; i >= -8; i--) {
    s = safe_add(s, i);
    t = safe_div(t, i);
  }
  printf("Result: %d,%d\n", s,t);
}
```

Imposed Constraints

- Generated programs never contains certain expressions/ statements, e.g. a naked addition of signed integers
- As a result, some code optimisations never trigger!

```
int main()
{
  int s = 5;
  int t = 2147483646;
  for (int i = 8; i >= -8; i--) {
    s = safe_add(s, i);
    t = safe_div(t, i);
  }
  printf("Result: %d,%d\n", s,t);
}
```

- **New fuzzer?** Compilers not yet immune to it but takes long time to develop!
- Can we relax the constraints imposed by existing fuzzers and find more bugs?

CsmithEdge: Be Less Conservative

- Get **closer to the edge** of the language semantic by being less conservative about undefined behaviour
- Modify Csmith to create more interesting programs by weakening the constraints **related to UB avoidance**
    1) Weaken generation constraints
    2) Remove unnecessary safe math wrappers
- The more diverse relaxed programs can be used directly to find crashes
- To find miscompilations, we use sanitizers to detect any UB introduced

Weaken Generation Constraints

These constraints guard against

- Dangling pointers
- Null pointer dereference
- Out-of-bound accesses
- Uninitialised accesses
- ...

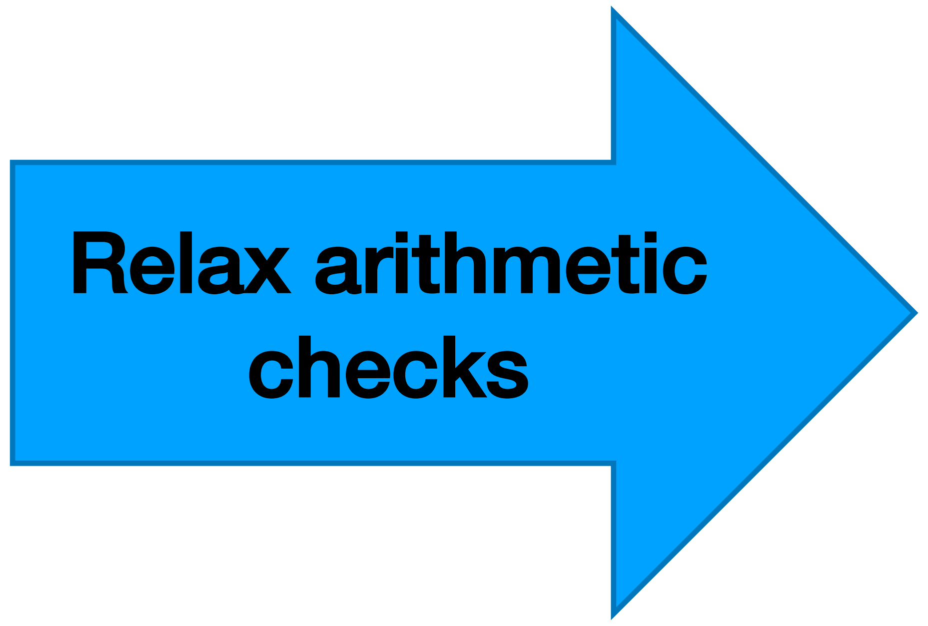
- Use a set of probabilities to decide separately per generated program:
  (1) A subset of constraints to weaken
  (2) The probability with each constraint type is weakened

- **Example**:
  - Allow null pointer deference 10% of the time and
  - Allow array out-of-bound accesses 25% of the time

Remove Unnecessary Safe Math Wrappers

CsmithEdge's dynamic analysis detects and replaces unnecessary safe math wrappers with the corresponding arithmetic operator

```c
int main()
{
  int s = 5;
  int t = 2147483646;
  for (int i = 8; i >= -8; i--) {
    s = safe_add(s, i);
    t = safe_div(t, i);
  }
  printf("Result: %d,%d\n", s,t);
}
```




```c
int main()
{
  int s = 5;
  int t = 2147483646;
  for (int i = 8; i >= -8; i--) {
    s = s+i;
    t = safe_div(t, i);
  }
  printf("Result: %d,%d\n", s,t);
}
```

CsmithEdge Evaluation in the Wild

Applied the tool regularly during development:

- 9 bugs, 7 previously unknown (5 fixed) + 2 independently reported:
  - 7 in GCC, 1 in LLVM, 1 in Visual Studio,
  - And several bugs in older compiler versions
  - All bugs were out-of-reach for Csmith

- Each bug required a different subset of relaxations to be discovered

```
int main(){
    const long ONE = 1L;
    long y = 0L;
    long x = ((long) (ONE || (y = 1L)) % 8L);
    printf("x = %ld, y = %ld\n", x, y);
}
```

Bug: violation of the short-circuiting rule
Buggy compiler version incorrectly evaluates second operand to || and prints 1

Required a naked %

C Compiler Fuzzing

<u>Blackbox Fuzzing</u>	<u>Greybox Fuzzing</u>	<u>Whitebox Fuzzing</u>
Highly successful, but starting to saturate	**Limited success, finding mostly front-end bugs**	**None available**
	?	

Greybox Fuzzing and Compiler Testing

- Greybox fuzzing highly successful for testing general software
  - E.g., Google found ~9k vulnerabilities and 28K bugs in 850 projects
  - More agile than blackbox fuzzing: lacking feedback, the latter saturates
- Why are standard greybox fuzzing tools ineffective for compiler testing?
  - Byte-level mutations likely to result in invalid programs!
  - Would only exercise the front-end

Greybox Fuzzing and Compiler Testing

- Mutation-based testing approaches (sometimes including coverage guidance) successful for dynamic languages
  - Dynamic languages more tolerant to code mutations (i.e. mutations less likely to result in invalid programs)
  - Front-end bugs often as valuable in the context of web security
  - LangFuzz (JavaScript/PHP), Superion (JavaScript), Nautilus (JavaScript, Lua, PHP, Ruby)

- Attempts for static languages include keyword dictionaries, protobuf descriptions of PL structure, regular expressions for common PL patterns
  - Still produce a high rate of invalid programs
  - Clang-Proto-Fuzzer: “Bugs are being fixed too slow (if at all)”
  - No-fuss Compiler Fuzzing: "*code that crashes a C or C++ compiler, but that includes extensive undefined behaviour may well be ignored by developers*”

GrayC

- Greybox fuzzing for testing compilers for C, representative of languages with lots of UB
- Pronounced “Grace”, in honor of compiler pioneer Grace Hopper
- Key idea: semantic-aware mutations which
  - Operate at the level of ASTs
  - Can both modify individual programs or combine existing ones
  - Have a configurable level of aggressiveness (likelihood of generating valid programs)
- Uses LibFuzzer as the underlying greybox fuzzing engine

Smithsonian Institution - Flickr: Grace Hopper and UNIVAC, CC BY 2.0

GrayC Mutators: Examples

- Duplicate a statement, delete a statement, delete a sub-expression, change the type of an expression
- Combine the body of a function with another function with the same number of arguments, either by concatenating bodies or interleaving their statements.

Example: Mutating Individual Programs

```c
typedef struct {
    unsigned w[3];
} Y;
Y arr[32];
int main() {
   int i=0;
   unsigned x=0;
   for (i=0; i<32; ++i)
       arr[i].w[1]=i == 1;
   for (i=0; i<32; ++i)
       x+=arr[1].w[1];
   if (x!=32)
       abort();
   return 0;
}
```

Delete statement →

```c
typedef struct {
    unsigned w[3];
} Y;
Y arr[32];
int main() {
   int i=0;
   unsigned x=0;
   for (i=0; i<32; ++i)
    for (i=0; i<32; ++i)
       x+=arr[1].w[1];
   if (x!=32)
       abort();
   return 0;
}
```

Duplicate statement →

```c
typedef struct {
    unsigned w[3];
} Y;
Y arr[32];
int main() {
   int i=0;
   unsigned x=0;
   for (i=0; i<32; ++i)
    for (i=0; i<32; ++i)
       x+=arr[1].w[1];
   x+=arr[1].w[1];
   if (x!=32)
       abort();
   return 0;
}
```

Example: Combining Programs

Combine the body of a function with another function with the same number of arguments, either by concatenating bodies or interleaving their statements

```
int dest_func(int x_dest,  int y_dest) {
   int b_dest = x_dest * y_dest;
   b_dest = b_dest + 5;
   return b_dest;
}
```

+

```
int src_func(int j_src,  int k_src){
   int m_src = j_src + k_src;
   return m_src;
}
```

Initialize variables corresponding to the src function to the args of dest function

```
int dest_func(int x_dest,  int y_dest) {
   int j_src = x_dest; int k_src = x_dest;
   int m_src = j_src + k_src;
   int b_dest = x_dest * y_dest;
   b_dest = b_dest + 5;
   return b_dest;
}
```

Interleave statements from src function

Randomly select return from src or dest

GrayC Aggresiveness

- Grayc has two modes:
  - Conservative mode
  - Aggressive mode

- Conservative mode has extra checks to ensure program validity, e.g.:
  - Change integer type to another integer type
  - Never replace array index with negative constant
  - Combine only functions with matching parameter types
  - etc.

GrayC Evaluation in the Wild

	Previously-unknown		Independently-reported	
	Confirmed	Fixed	Confirmed	Fixed
GCC	8	8	3	3
LLVM	2	2	1	0
MSVC	3	1	0	0
Frama-C	11	11	1	1
TOTAL	24	22	5	4

29 Bugs

GrayC Bugs: Compiler Component

	Front-end	Middle-/Back-end
GCC	2	9
LLVM	1	2
MSVC	3	0
Frama-C	2	10
TOTAL	8	21

29 Bugs

GrayC Controlled Experiments

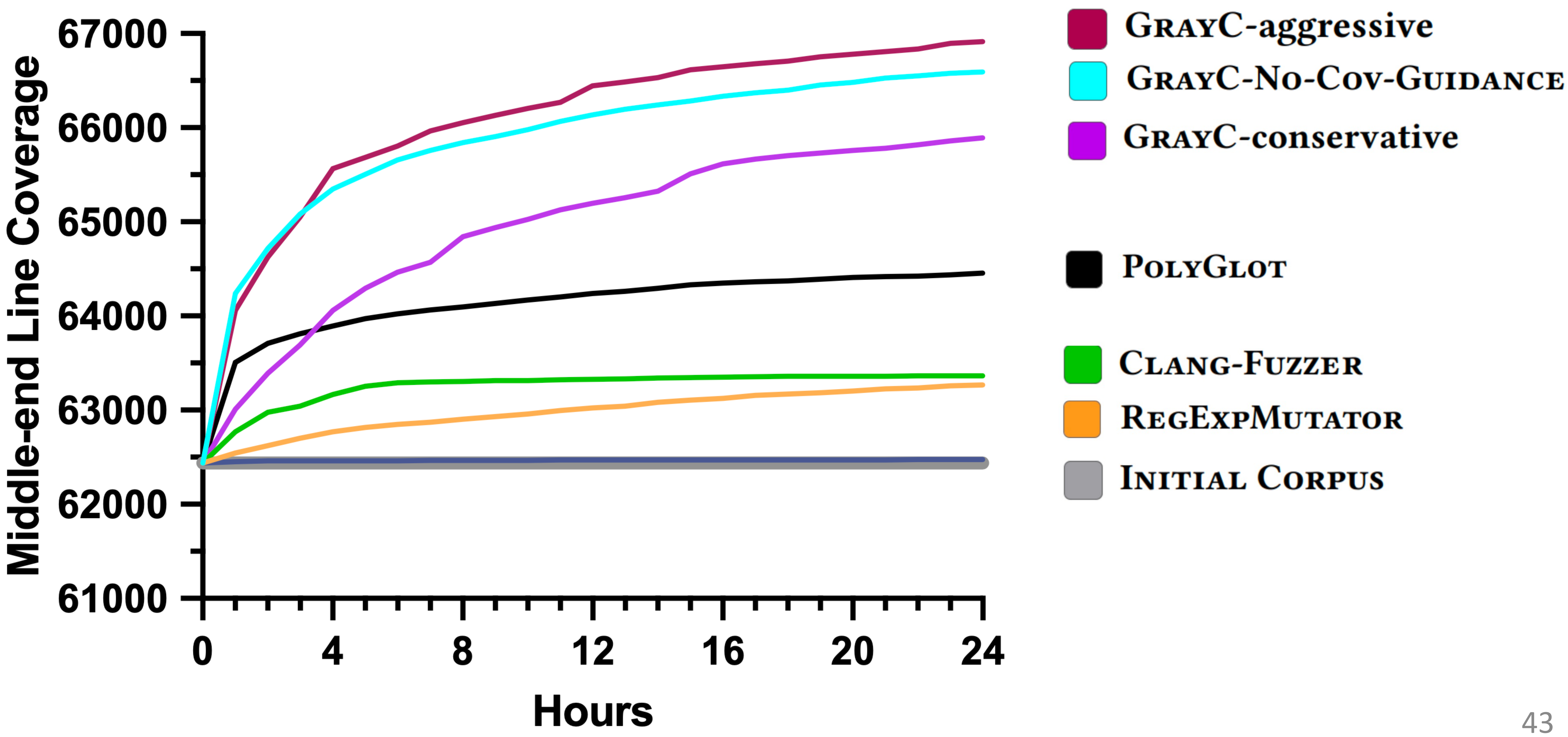
1)	GrayC-Aggressive	Default GrayC
2)	GrayC-Conservative	Do the extra checks have an impact?
3)	GrayC-No-Cov-Guidance	Does coverage guidance matter?
4)	GrayC-Fragments-Fuzzing	Only code fragments injection, no coverage (similar to LangFuzz)
5)	Clang-Fuzzer	Greybox fuzzing with byte-level mutations
6)	Csmith	Grammar-based fuzzing
7)	Grammarinator	Grammar-based fuzzing (ANTLR C grammar)
8)	PolyGlot	Language-agnostic AFL-based fuzzer, based on semantic error fixing
9)	RegExpMutator	LibFuzzer-based fuzzer based that uses regexp-based mutations

24h per tool, 10 repetitions

Throughput & Static Validity

	Programs/h	Statically-valid (%)
Csmith	1,144	99.96%
GrayC-conservative	1,691	99.69%
GrayC-aggressive	2,906	99.47%
GrayC-Fragments-Fuzzing	3,957	99.08%
PolyGlot	714	91.20%
GrayC-No-Cov-Guidance	4,700	75.41%
RegExpMutator	1,390	19.1%
Clang-Fuzzer	1,183	1.55%
Grammarinator	5,391	0.0%

Middle-End Coverage in LLVM



Bugs Found

Tool	Component		Fix Rate
	Middle	Front	
GrayC-aggressive	6	-	100%
GrayC-No-Cov-Guidance	4	-	100%
GrayC-conservative	2	-	100%
RegExpMutator	2	1	67%
Clang-Fuzzer	1	4	60%
PolyGlot	-	1	0%
Csmith	-	-	0%
Grammarinator	-	-	0%
GrayC-Fragments-Fuzzing	-	-	0%

Conservative vs Aggressive Mode

- Not enforcing strict checks to maximize validity seems to be better
- Two possible explanations

1) Some restrictions not needed and lead to less diverse programs
   - Similar to the Csmith vs CsmithEdge story

2) Some bugs and extra coverage triggered by ”almost valid” programs

Conservative vs Aggressive Mode

- Valid programs needed for finding miscompilations and contributing test cases to compilers' regression suites
- We contributed **30** test cases to LLVM's test suite with **23** already accepted

CsmithEdge

- Blackbox compiler fuzzers tend to saturate over time
- One key limitation of existing compiler fuzzers is that they produced overly restrictive programs
- Relaxing these restrictions can extend their reach
- CsmithEdge found 9 new bugs, all of which were out-of-reach for Csmith

GrayC

- Greybox compiler fuzzing for languages with extensive UB is feasible
- Key idea is to use AST-level semantics-aware mutations
- GrayC found 29 bugs (26 fixed), with 24 previously unknown (22 fixed)
- Significant gains in terms of bug-finding & coverage compare to prior work
- We used GrayC to contribute 30 test cases (23 accepted) to the LLVM compiler