Bug Finding in Compiler Toolchains

Giuseppe Antonio Di Luna, Fiorella Artuso
diluna@diag.uniroma1.it
artuso@diag.uniroma1.it

RoadMap

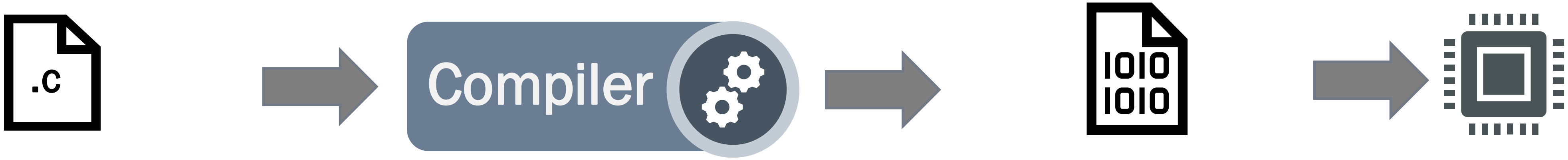
Preliminaries on compiler toolchain (compiler and debugger) together with problems connected with internal bugs.

First Homework Assignment

The Compiler Toolchain

Source code in C language

Executable code in machine language

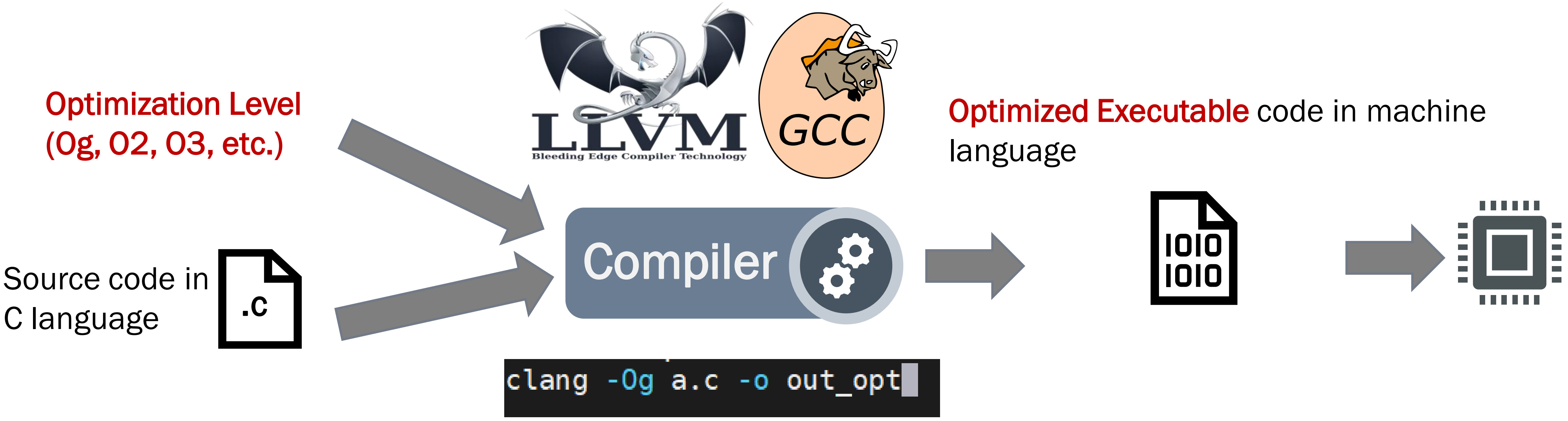



```
root@e1e844747c68:/home/stepping# cat -n a.c
     1  int f(int x) {
     2      int a;
     3      for(int i=0; i<5; i++)
     4          a *= x+2;
     5      return a;
     6  }
     7
     8  int main(){
     9          f(3);
    10  }
```

```
clang a.c -o out
```

```
root@e1e844747c68:/home/stepping# file out
out: ELF 64-bit LSB executable, x86-64, ver
sion 1 (SYSV), dynamically linked, interpre
ter /lib64/ld-linux-x86-64.so.2, for GNU/Li
nux 3.2.0, with debug_info, not stripped
```

The Compiler Toolchain - Optimizations



- **Most** of the **software** running in **production** is produced by an **optimizing compiler.**
- The role of **optimizations** is to apply a **series of trasformations** to output a program which is **semantically equivalent** to the original one but it uses **fewer resources** and it is **faster** to execute.

The Compiler Toolchain - Optimizations

```
root@e1e844747c68:/home/stepping# cat -n a.c
     1  int f( int x) {
     2      int a;
     3      for( int i=0; i<5; i++)
     4          a *= x+2;
     5      return a;
     6  }
     7
     8  int main(){
     9          f(3);
    10  }
```

No optimizations applied

```
0000000000400480 <f>:
  400480:       55                      push   %rbp
  400481:       48 89 e5                mov    %rsp,%rbp
  400484:       89 7d fc                mov    %edi,-0x4(%rbp)
  400487:       c7 45 f4 00 00 00 00    movl   $0x0,-0xc(%rbp)
  40048e:       83 7d f4 05             cmpl   $0x5,-0xc(%rbp)
  400492:       0f 8d 1b 00 00 00       jge    4004b3 <f+0x33>
  400498:       8b 45 fc                mov    -0x4(%rbp),%eax
  40049b:       83 c0 02                add    $0x2,%eax
  40049e:       0f af 45 f8             imul   -0x8(%rbp),%eax
  4004a2:       89 45 f8                mov    %eax,-0x8(%rbp)
  4004a5:       8b 45 f4                mov    -0xc(%rbp),%eax
  4004a8:       83 c0 01                add    $0x1,%eax
  4004ab:       89 45 f4                mov    %eax,-0xc(%rbp)
  4004ae:       e9 db ff ff ff          jmpq   40048e <f+0xe>
  4004b3:       8b 45 f8                mov    -0x8(%rbp),%eax
  4004b6:       5d                      pop    %rbp
  4004b7:       c3                      retq
  4004b8:       0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
  4004bf:       00
```

The Compiler Toolchain - Optimizations

```
root@e1e844747c68:/home/stepping# cat -n a.c
     1  int f(int x) {
     2          int a;
     3          for(int i=0; i<5; i++)
     4                  a *= x+2;
     5          return a;
     6  }
     7
     8  int main(){
     9                  f(3);
    10  }
```

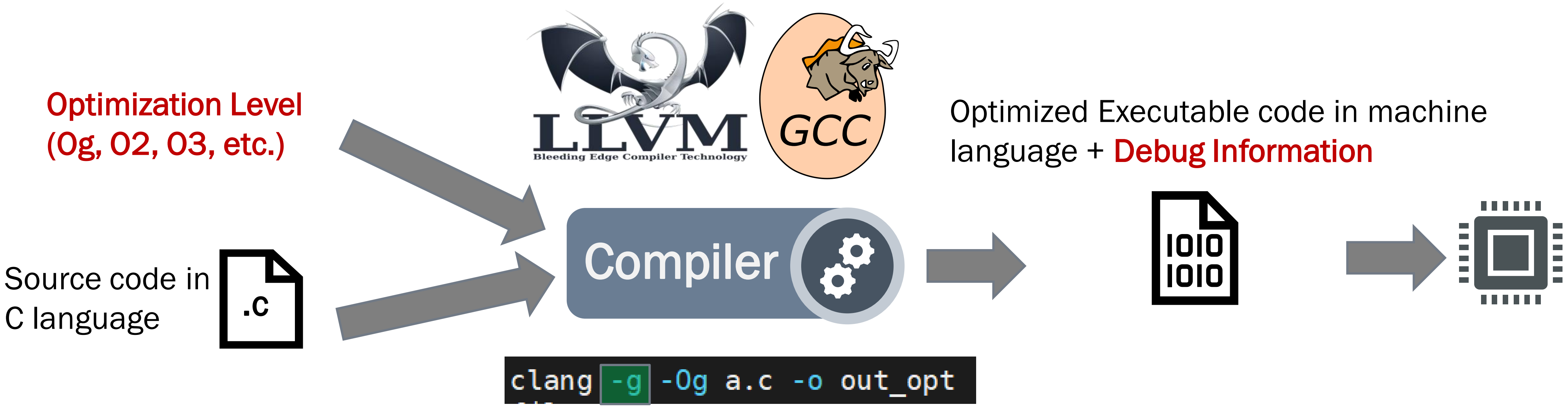
Loop Invariant Code Motion

```
0000000000400480 <f>:
  400480:       83 c7 02                add    $0x2,%edi
  400483:       b9 05 00 00 00          mov    $0x5,%ecx
  400488:       0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
  40048f:       00
  400490:       0f                      imul   %edi,%eax
  400493:       83 c1 ff                add    $0xffffffff,%ecx
  400496:       75                      jne    400490 <f+0x10>
  400498:       c3                      retq
  400499:       0f 1f 80 00 00 00 00    nopl   0x0(%rax)
```

Source Code optimized

```
root@e1e844747c68:/home/stepping# cat -n a_opt.c
     1  int f(int x) {
     2          int a, z=x+2;
     3          for(int i=0; i<5; i++)
     4                  a *= z;
     5          return a;
     6  }
     7
     8  int main(){
     9                  f(3);
    10  }
```

The Compiler Toolchain - Debugger



- Debug information is **generated by compilers** and can be stored either inside object files or inside separate files which encode debugging data formats (e.g. DWARF).

- This information is usually **consumed by debugger programs** to gain access to high level information from the source code of the binary (such as the variable values, backtrace information, executed source line) so as to provide **support** to humans **developers** to more easily **detect errors.**

The Compiler Toolchain - Debugger

Debug
Execution Trace

```
root@e1e844747c68:/home/stepping# cat -n a.c
     1  int f( int x) {
     2      int a;
     3      for( int i=0; i<5; i++)
     4          a *= x+2;
     5      return a;
     6  }
     7
     8  int main(){
     9          f(3);
    10  }
```

```
root@e1e844747c68:/home/stepping# lldb out
(lldb) target create "out"
Current executab                                  (x86_64).
(lldb) b main
Breakpoint 1: where = out`main + 4 at a.c:9:9, address =
(lldb) r
Process 181                    g/out' (x86_64)
Process 181 stopped
* thread #1, name = 'out', stop reason = breakpoint 1.1
    frame #0: 0x00000000004004c4 out`main at a.c:9:9
   6    }
   7
   8    int main(){
-> 9            f(3);
   10   }
```

Set the breakpoint on main

Launch the execution

The Compiler Toolchain - Debugger


```
root@e1e844747c68:/home/stepping# cat -n a.c
     1	int f(int x) {
     2		int a;
     3		for(int i=0; i<5; i++)
     4			a *= x+2;
     5		return a;
     6	}
     7
     8	int main(){
     9		f(3);
    10	}
```

```
(lldb) di -l

-> 8      int main(){
-> 9              f(3);

out`main:
->  0x4004c4 <+4>: movl   $0x3, %edi
    0x4004c9 <+9>: callq  0x400480                  ; f at a.c:1
(lldb) s
Process 181 stop
* thread #1, name = 'out', stop reason = step in
    frame #0: 0x0000000000400487 out`f(x=3) at a.c:3:13
   1      int f(int x) {
   2          int a;
-> 3          for(int i=0; i<5; i++)
   4              a *= (x+2);
   5          return a;
   6      }
   7
```

See the corresponding assembly instruction

Make a step on the next source instruction

The Compiler Toolchain - Debugger

Debug
Execution Trace

```
root@e1e844747c68:/home/stepping# cat -n a.c
     1  int f(int x) {
     2      int a;
     3      for(int i=0; i<5; i++)
     4          a *= x+2;
     5      return a;
     6  }
     7
     8  int main(){
     9          f(3);
    10  }
```

```
(lldb) p i
(int) $3 = 0
(lldb) p x
(int) $4 = 3
(lldb) bt
* thread #1, name = 'out', stop reason = step in
  * frame #0: 0x0000000000400487 out`f(x=3) at a.c:3:13
    frame #1: 0x00000000004004ce out`main at a.c:9:9
    frame #2: 0x00007ffff7a03bf7 libc.so.6`__libc_start_m
    frame #3: 0x00000000004003ba out`_start + 42
```

See variable values

Check backtrace information

The Compiler Toolchain May Contain Bugs

- Compilers are long-lasting and widely-used software. However, since they are written by humans, they are **not bug-free**.

- For example, more than **3k internal bugs** have been found in GCC since it is created.

- Compiler bugs can:
  - result in **unintended program executions** leading to catastrophic consequences in **security-sensitive** applications.
  - They can also have a negative impact on **developer productivity** in debugging a program (Where is the bug?).

- It is critical to **improve the compiler correctness.** They are big and complex software (for instance, GCC is around 15 million lines of code), thus it is not easy to test them.

The Compiler Toolchain May Contain Bugs

Bugzilla – Bug List

Home | New | Browse | Search | Search | [?] | Reports | Help | Log In | Forgot Password

New user self-registration is disabled due to spam. For an account please email bugs-admin@lists.llvm.org with your e-mail address and full name.

Tue Oct 19 2021 09:03:23 PDT
Go ahead, compile my source.

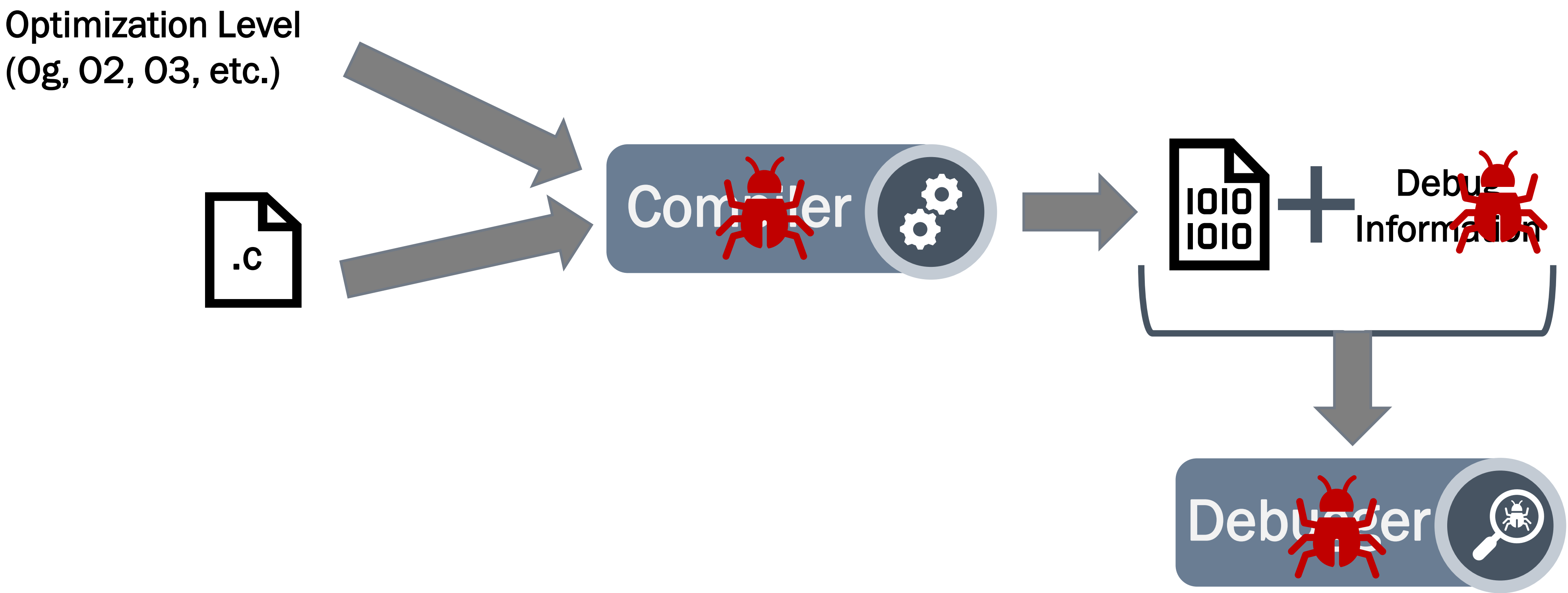
Hide Search Description

Status: RESOLVED **Product:** clang

This result was limited to 500 bugs. See all search results for this query.

ID	Product	Comp	Assignee	Status	Resolution	Summary	Changed
31504	clang	C++	bruno.cardoso	RESO	FIXE	REGRESSION: Including <float.h> causes build failures on darwin when darwin's /usr/include/float.h is present (pre-Lion)	2017-08-01
47184	clang	Driver	unassignedclangbugs	RESO	FIXE	Merge cd5ab56bc406c3f9a6f593f98c63dafb53547ab1 into 11.0	2020-08-18
22293	clang	-New Bug	unassignedclangbugs	RESO	FIXE	Clang 3.6 asserting: UNREACHABLE executed lib/IR/Value.cpp:781!	2015-02-12
42011	clang	LLVM Cod	rnk	RESO	FIXE	merge r359809 into 8.0	2019-06-26
37457	clang	Frontend	unassignedclangbugs	RESO	DUPL	Use of _Atomic() now fails with address argument to atomic operation must be a pointer to a trivially-copyable type	2018-08-15
13015	clang	-New Bug	unassignedclangbugs	RESO	FIXE	missing destructor call for temporary created by C++11 braced initializer list syntax as function argument	2013-04-05
20629	clang	-New Bug	unassignedclangbugs	RESO	WORK	clang fails to build compiler-rt (sanitizer-allocator.cc)	2015-11-30
32673	clang	C++17	unassignedclangbugs	RESO	FIXE	Segmentation fault with class template argument deduction and variadic template class with variadic constructor and variadic inheritance	2017-08-15

The Compiler Toolchain: Where are Bugs?



A Compiler Bug - CVE-2019-15847

```
GNU nano 4.3
#include <stdio.h>
#include <stdint.h>

int main(){
uint64_t darn[32];
for(size_t i=0; i!=32; ++i)
        darn[i] = __builtin_darn();
for(size_t i=0; i!=32; ++i)
        printf("%016lX\n", darn[i]);
}
```

https://nvd.nist.gov/vuln/detail/CVE-2019-15847

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=91481

```
./out
32A7727F89101CF3
32A7727F89101CF3
32A7727F89101CF3
32A7727F89101CF3
32A7727F89101CF3
32A7727F89101CF3
32A7727F89101CF3
32A7727F89101CF3
32A7727F89101CF3
32A7727F89101CF3
32A7727F89101CF3
32A7727F89101CF3
32A7727F89101CF3
32A7727F89101CF3
```

The POWER9 ISA includes a built-in function for **hardware random number generator** (__builtin_darn()).

If you have a program calling multiple time that function and you compile that program on a **POWER9 backend with GCC before version 10**, it will produce always the **same results**.

This is because **the compiler optimize multiple calls** of the __builtin_darn into a single call, thus **reducing the entropy** of the random number generator.

This did not happen if optimizations were disabled.

A Debug Information Bug

a.c

```c
int  a,  b,  c;
int  main()
{
   {int  ui1  =  5,  ui2  =  b
      ;
     c  =
       ui2  ==  0  ?
       ui1  :
       (ui1  /  ui2);
   }
}
```

```
$ clang -g –Og a.c –o opt
$ lldb opt
(lldb) target create "opt"
Current executable set to 'opt' (x86_64).
(lldb) b main
Breakpoint 1: where = opt`main at a.c:4:26,
(lldb) r
Process 65 launched: 'opt' (x86_64)
1    int a,  b,  c ;
  2    int main ()
  3    {
-> 4        {int ui1 = 5,  ui2 = b;
  5           c =
  6              ui2 == 0 ?
  7              ui1 :
```

A Debug Information Bug

a.c

```c
int a, b, c;
int main()
{
  {int ui1 = 5, ui2 = b
     ;
    c =
      ui2 == 0 ?
      ui1 :
      (ui1 / ui2);
  }
}
```

```
(lldb) s
Process 65 stopped
   5        c =
   6          ui2 == 0 ?
   7          ui1 :
-> 8          (ui1 / ui2);
   9        }
   10   }
(lldb) di
opt`main:
   0x400480 <+0>:  movl   0x200ba6(%rip), %ecx      ; b
-> 0x400486 <+6>:  movl   $0x5, %eax
   0x40048b <+11>: testl  %ecx, %ecx
```

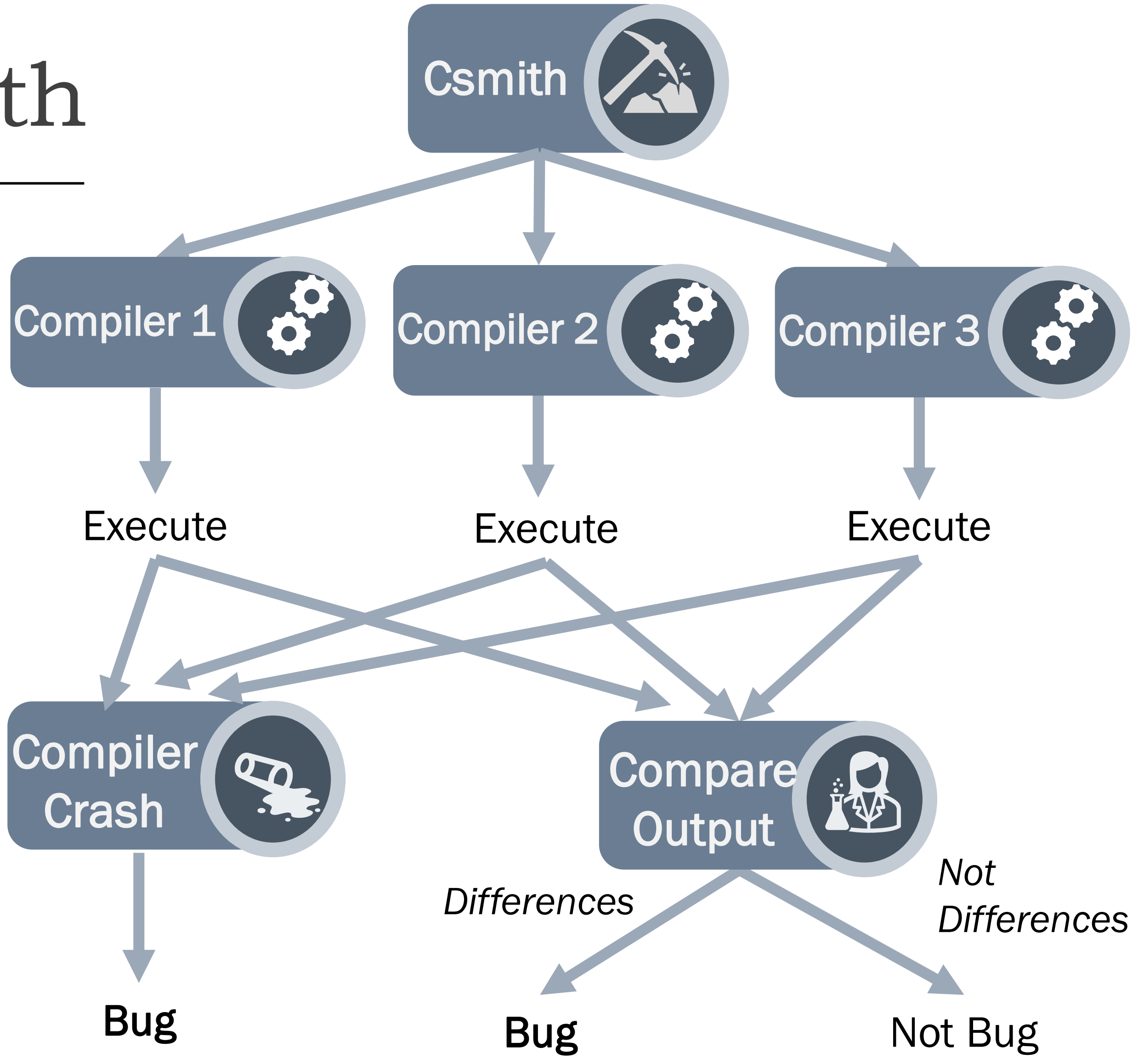
Stepping on Dead Code

This is **a compiler bug** and it is caused by a compiler optimization. In fact, optimization passes can move instructions across basic blocks without properly updating the corresponding debug information.

Compiler Testing - CSmith

- **Randomized Testing (Fuzzing):** It is a black-box testing technique that consists in feeding a program with random inputs. **Csmith** is a Random C program Generator.

- **Differential Testing:** It is based on the assumption that different implementations of the same specification should always produce the same result. When different results are produced one the implementation should be faulty.



"Finding and Understanding Bugs in C Compilers". Yang et al. PLDI'11

ML-Based Compiler Testing

"DeepFuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing" Liu, Xiao et al. AAAI '19

Challenge

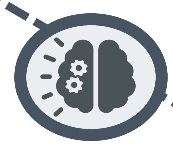
In order to generate random C-programs it is necessary to **manually** build a grammar-based engine and it is a **daunting** task (ISO/IEC 9899:2011 has 696 pages of detailed specifications!)

Proposed Solution: DeepFuzz

Train a **machine learning model** to learn the "grammar" from "real code" and produce new C programs to fuzz compilers. Given a certain sequence of source code, this model should be able to predict which line comes next.

```
int a;
char b;
char *c[2];
void func_1() {
    int l_5 = 3, d = 0;
    for (; d < 2; d++)
    for (;;) {
        if (l_5)
            break;
        a = &l_5;
        __assert_fail();
    }
}
int main() { func_1();}
```

Original C Program



```
int a;
char b;
char *c[2];
void func_1() {
    int l_5 = 3, d = 0;
    for (; d < 2; d++)
    c[d] = b;
    for (;;) {
        if (l_5)
            break;
        a = &l_5;
        l_5 +=2
        __assert_fail();
    }
}
int main() { func_1(); }
```

New Generated C Program

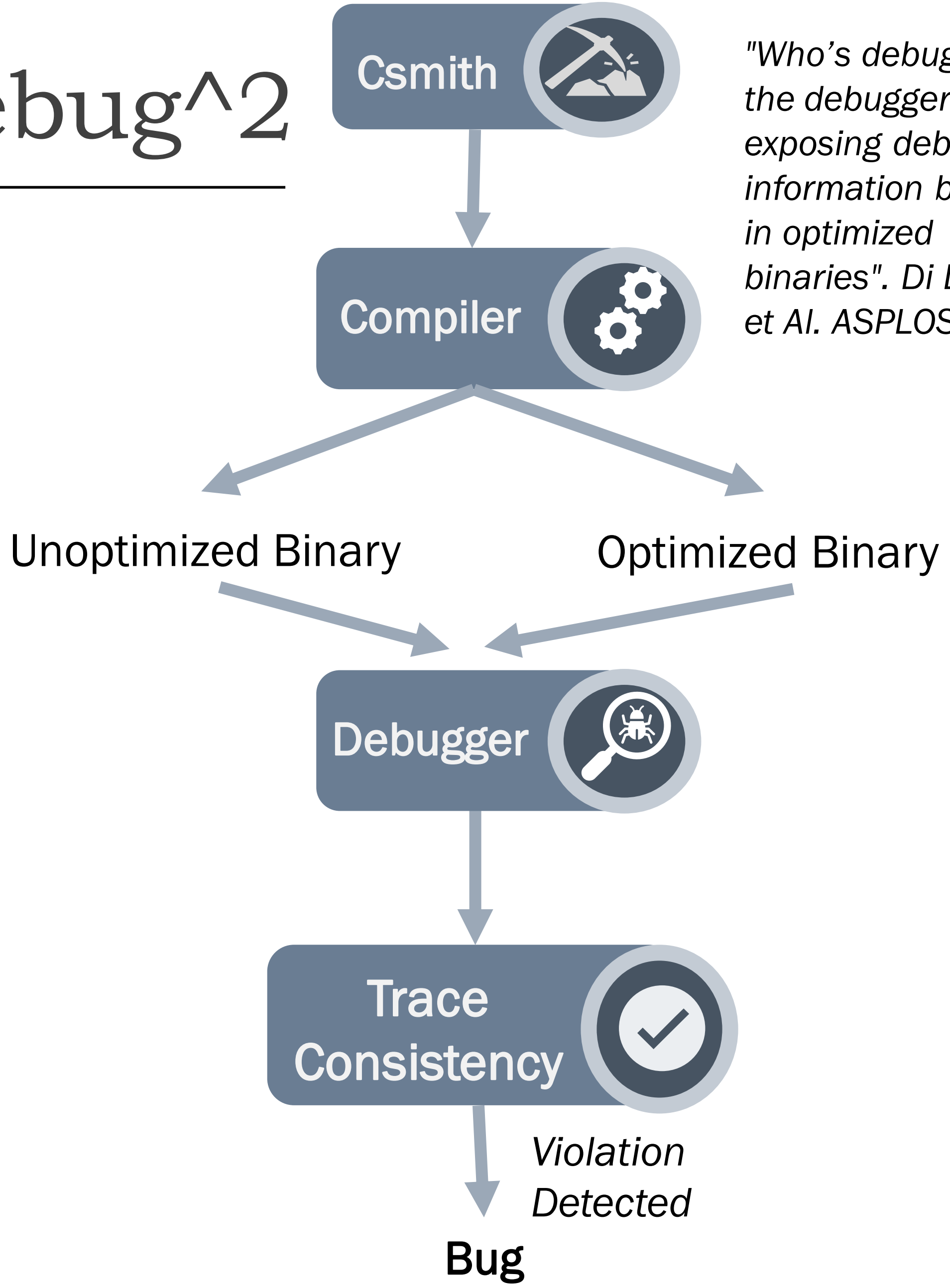
Debug Information Testing – Debug^2

Problem

Preserving the correctness of debug information while optimization passes are applied is an extremely complex task.

Solution: Differential testing

- Generate a random program using a random program generator (e.g. csmith).
- Compile the program both without and with optimizations.
- Use the debugger to obtain the corresponding execution traces.
- Check for inconsistencies inside optimized and unoptimized trace (e.g. a source line appears to be executed in the optimized trace while the unoptimized trace never executes it)



"Who's debugging the debuggers? exposing debug information bugs in optimized binaries". Di Luna et Al. ASPLOS'20

RoadMap

First Homework Assignment

Preliminaries on compiler toolchain (compiler and debugger) together with problems connected with internal bugs.

Debug Information Testing with Machine Learning

HOMEWORK

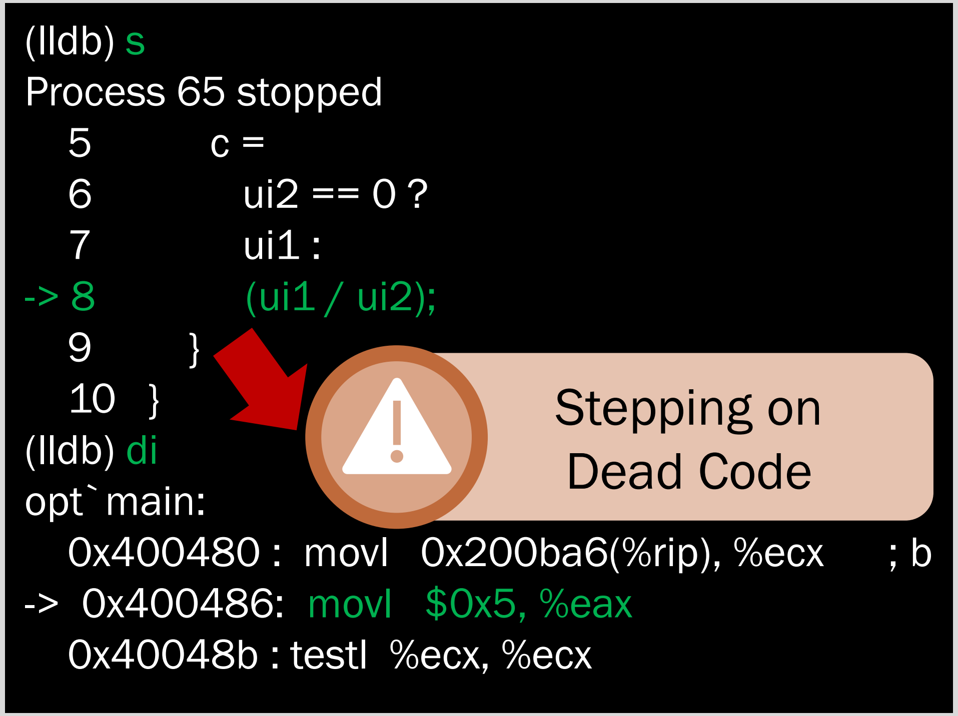
Debugging Debug Information with Neural Networks

Debug^2 : Non-ML-Based existing Solution

based on differential approaches in which debug information of unoptimized and optimized binaries is compared searching for manually defined inconsistencies.

ML-Based Solution

Leverage Machine Learning to automatically discover incorrect debug information included in optimized binaries.



Preliminary Definitions

```
typedef short int  int16_t;
typedef int  int32_t;
int16_t g_4;
int32_t g_2, g_3 = 4;
int32_t *g_6 = &g_2;
void func_3() {
        int32_t **l_1876 = &g_6;
        int32_t l_23 = 1;
        if (g_3) {
                int32_t *l_2441 = g_4;
                (**l_1876) = l_2441;
        }
        *l_1876 = &l_23;
        }
int main() { func_3(); }
```

Mapping Assembly to source code

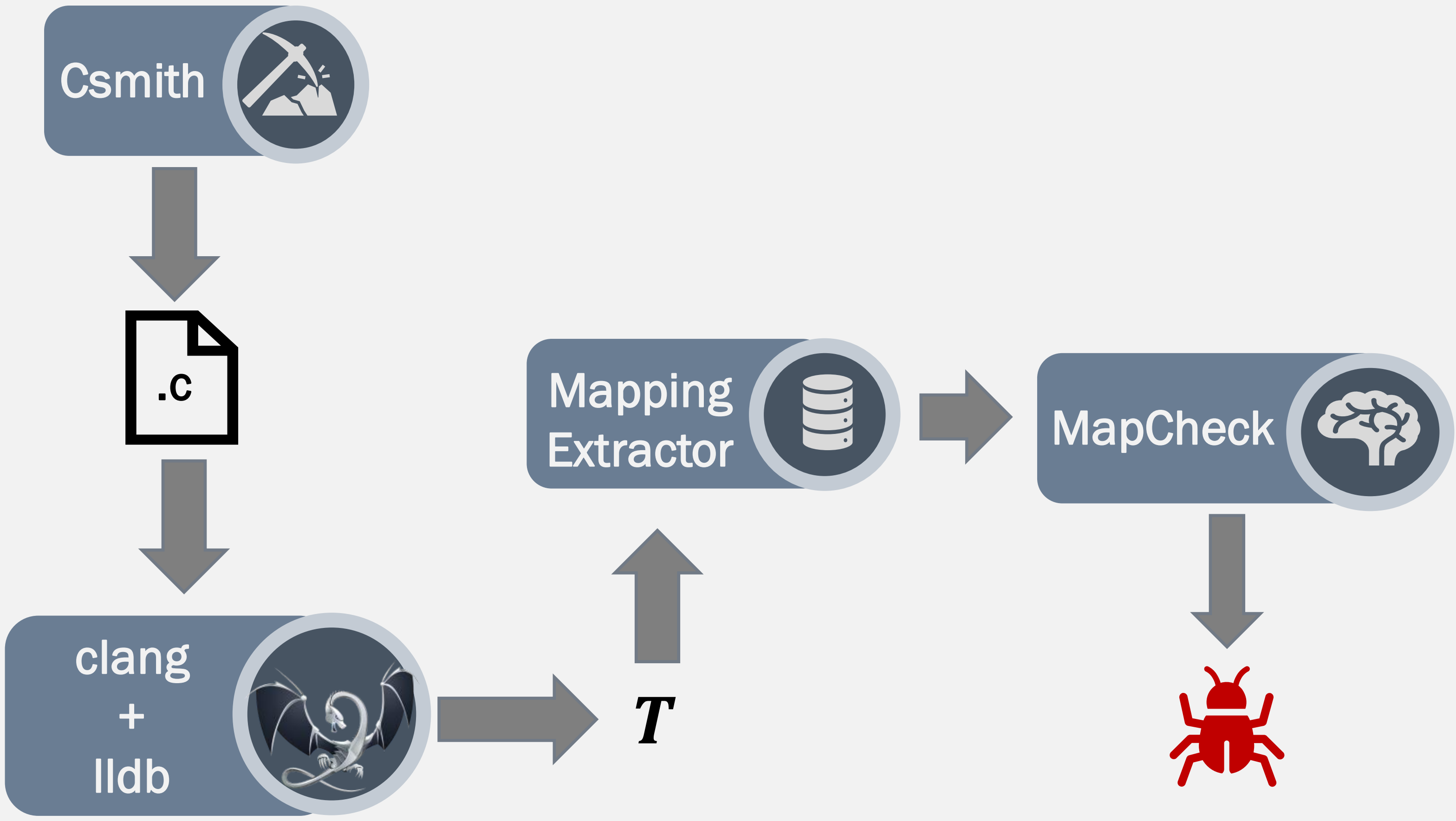
callq 0x401110	int main() { func_3(); }
movl $0x1, -0x4(%rsp)	int32_t l_23 = 1;
cmpl $0x0, 0x2f09(%rip), je 0x401131	if (g_3) {
...	

An **debug execution trace** T: $[s_0, s_1, \dots, s_n]$ is a **sequence of steps** obtained by setting the breakpoint on the entry point and by repeatedly **stepping** over **assembly instruction** untill the program exits.

Starting from this debug trace we define:

- **Mapping Assembly to source code:** for each line in the trace we have associated a sequence of assembly instructions (used to execute the line).

A Bug Detection System



What You Have…

We provide you with the dataset (**mapping_traces_00.csv**) which is a **pandas dataframe** (*tab separated csv file*) containing 100k samples. Each sample contains three type of information:

- **List of assembly instructions**
- **source line**
- **Label** indicating wether the mapping is correct (0) or incorrect (1)

```
import pandas as pd
public_dataset = pd.read_csv("/home/MLDebugSeminar/mapping_traces_00.csv", sep="\t")
```

```
public_dataset[["instructions","source_line","bug"]]
```

	instructions	source_line	bug
0	movl 11 MEM cmpl 36 MEM jne MEM	l 20 = (safe mod func uint 16 t u u ...	1
1	leaq l 25 4 %rax cmpq %rax l 25 3 je MEM jmp MEM	((l 25 3 = = & l 25 4 | | l 25 3 = ...	0
2	movl HIGHVAL l 46 43	uint32t l 46 43 = HIGHVAL ;	0
3	leaq l 33 87 %rax movq %rax l 33 86	const uint8t * * * l 33 86 = & l 33 87 ;	0
4	movb -1 l 10 59 9	uint32t l 17 91 = 9 ;	1
...	...	...	...

l_4643

What You Have…

We provide you with the dataset (**mapping_traces_00.csv**) which is a **pandas dataframe** (*tab separated csv file*) containing 100k samples. Each sample contains three type of information:

- **List of assembly instructions**
- **source line**
- **Label** indicating wether the mapping is correct (0) or incorrect (1)

```
import pandas as pd
public_dataset = pd.read_csv("/home/MLDebugSeminar/mapping_traces_00.csv", sep="\t")
```

```
public_dataset[["instructions","source_line","bug"]]
```

	instructions	source_line	bug
0	movl 11 MEM cmpl 36 MEM jne MEM	l 20 = (safe mod func uint 16 t u u ...	1
1	leaq l 25 4 %rax cmpq %rax l 25 3 je MEM jmp MEM	((l 25 3 = = & l 25 4 | | l 25 3 = ...	0
2	movl HIGHVAL l 46 43	uint32t l 46 43 = HIGHVAL ;	0
3	leaq l 33 87 %rax movq %rax l 33 86	const uint8t * * * l 33 86 = & l 33 87 ;	0
4	movb -1 l 10 59 9	uint32t l 17 91 = 9 ;	1
...	...	...	...

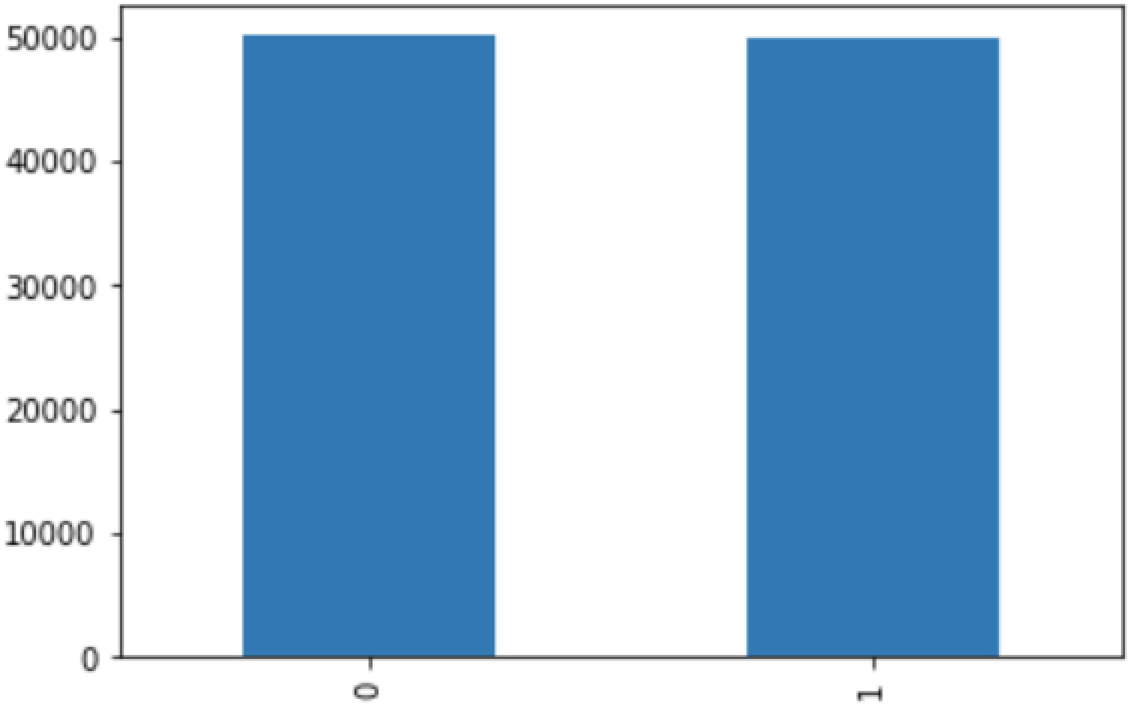
l_10599 l_1791

What You Have…

- **Duplicated** samples have already been **removed** from the dataset.
- Dataset is **balanced.**

```
public_dataset["bug"].value_counts().plot(kind='bar')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fabac22bc90>
```



What You Have…

Prepocessing and tokenization have already been applied.

Assembly instructions preprocessing

- All memory accesses have been substituted with a special token MEM,
- Immediate operands have been converted into base ten and have been substituted with special token ("HIGHVAL") if they are above a certain treshold (i.e. 1,000).
- We obtain a single string by concatenating all assembly instructions using whitespaces.

Source Line preprocessing

- We used the same strategy as before to preprocess numbers in the source line.
- We then used codeprep* to tokenize source line strings.

* Karampatsis, R. M.; Babii, H.; Robbes, R.; Sutton, C.; and Janes, A. 2020. Big Code != Big Vocabulary: OpenVocabulary Models for Source Code. ICSE '20.

What You Need To Do…

- Use **any classifier** that you have seen during the course to solve the following **binary classification problem**:

> Given a mapping assembly instructions – source line predict wether it contains a bug or not

- Use the mapping_trace_00.csv file to obtain training, validation and test set.
- We will provide you a **blind test set** (i.e. without ground truth labels) with 10k samples.
- Use **your trained model** to **predict** wether mapping pairs inside the blind test set are bugs or not.
- Send a **.txt file** with corresponding predicted labels: **one label for each row** in the **same order** as the one of the blind test set.

What You Need To Do…

- Use **any classifier** that you have seen during the course to solve the following **binary classification problem:**

Given a mapping assembly instructions … source line pre

- Use the mapping_tr… .
- We will provide you … samples.
- Use **your trained mo**… set are bugs or not.
- Send a **.txt file** with co… in the **same order** as the one of the blind test set.

Hint

Consider instructions and source line as a unique sequence before feature extraction.

Use one of the techniques for extracting features from text (e.g. bag of words, etc.). You can also try to use only mnemonics or the whole assembly instruction sequence.

Questions?