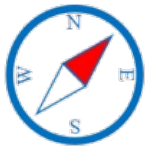
南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

香港科技大學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY



COMPASS Lab
COMPuter And System Security Lab

DTD: Comprehensive and Scalable Testing for Debuggers

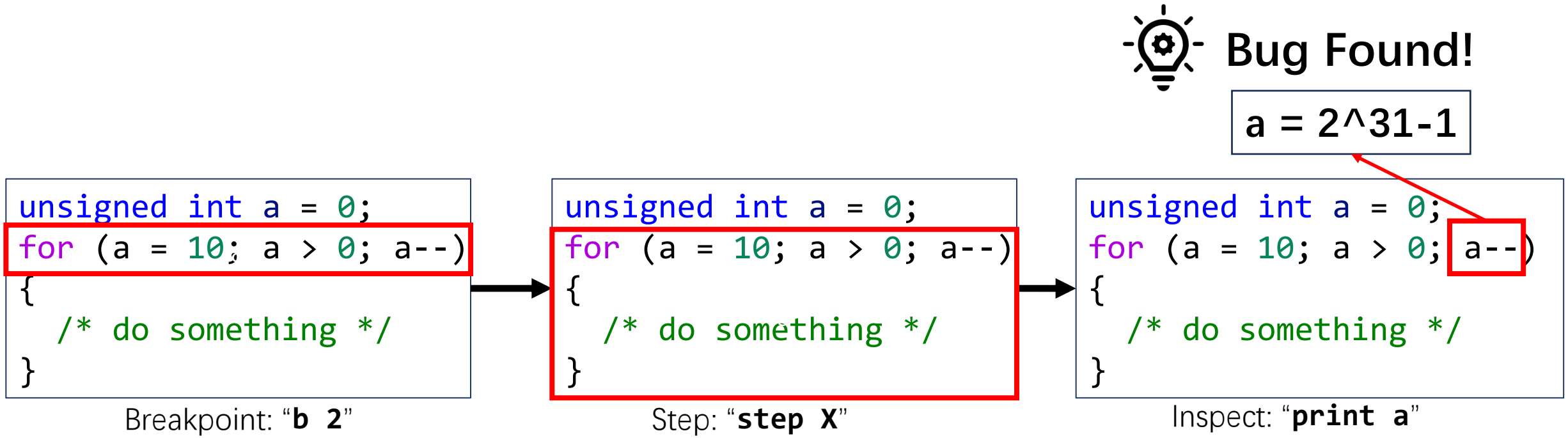
Hongyi Lu[1,2], Zhibo Liu[2,*], Shuai Wang[2], Fengwei Zhang[1,*]

[1]Southern University of Science and Technology

[2]Hong Kong University of Science and Technology

Background & Insight

Debuggers


Bug Found!
a = 2^31-1
unsigned int a = 0;
for (a = 10; a > 0; a--)
{
/* do something */
}
Breakpoint: "b 2"
unsigned int a = 0;
for (a = 10; a > 0; a--)
{
/* do something */
}
Step: "step X"
unsigned int a = 0;
for (a = 10; a > 0; a--)
{
/* do something */
}
Inspect: "print a"

DWARF

```
imul    -0x8(%rbp),%rax
mov     $0x3b9aca07,%edx
mov     %edx,%ecx
mov     $0x0,%edx
div     %rcx
mov     %rdx,-0x8(%rbp)
```

Executable:

[0x401123 – 0x401137]

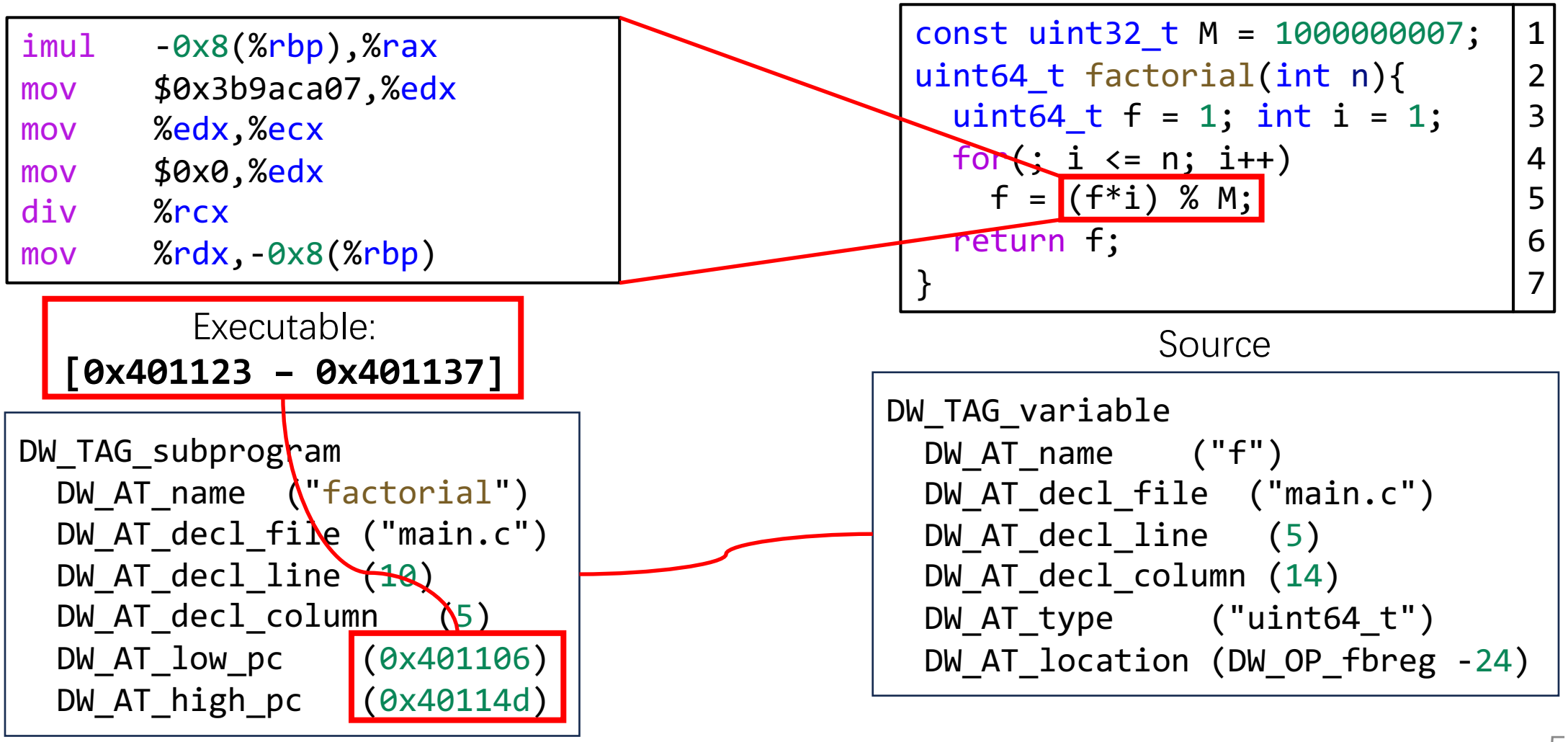
```
const uint32_t M = 1000000007;                  1
uint64_t factorial(int n){                      2
  uint64_t f = 1; int i = 1;                    3
  for(; i <= n; i++)                            4
    f = (f*i) % M;                              5
  return f;                                     6
}                                               7
```

Source

Functions:
Where is the source file?
What is the function name?

Variables:
What type?
How is its value computed?

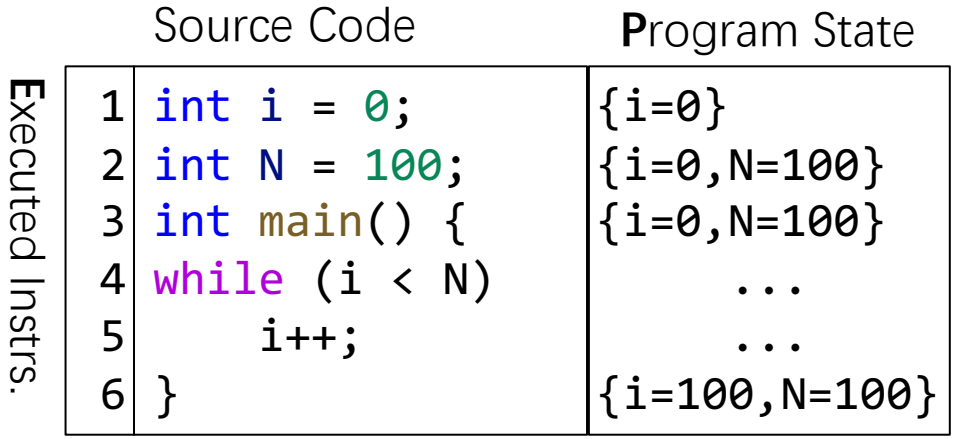
DWARF


imul -0x8(%rbp),%rax
mov $0x3b9aca07,%edx
mov %edx,%ecx
mov $0x0,%edx
div %rcx
mov %rdx,-0x8(%rbp)
const uint32_t M = 1000000007;
uint64_t factorial(int n){
uint64_t f = 1; int i = 1;
for(; i <= n; i++)
f = (f*i) % M;
return f;
}
1
2
3
4
5
6
7
Executable:
[0x401123 – 0x401137]
Source
DW_TAG_subprogram
DW_AT_name ("factorial")
DW_AT_decl_file ("main.c")
DW_AT_decl_line (10)
DW_AT_decl_column (5)
DW_AT_low_pc (0x401106)
DW_AT_high_pc (0x40114d)
DW_TAG_variable
DW_AT_name ("f")
DW_AT_decl_file ("main.c")
DW_AT_decl_line (5)
DW_AT_decl_column (14)
DW_AT_type ("uint64_t")
DW_AT_location (DW_OP_fbreg -24)

Debugger Testing Requirements

Three Types of Coverage

- **P**rogram State
- **E**xecuted Instruction
- **D**ebugger Information



Ideally, **P+E = D**, but it's very slow (>30 min).

We thus propose **P+WE = D**.

Weak**E**: Every **unique** instruction is covered.

Previous Works

They either:

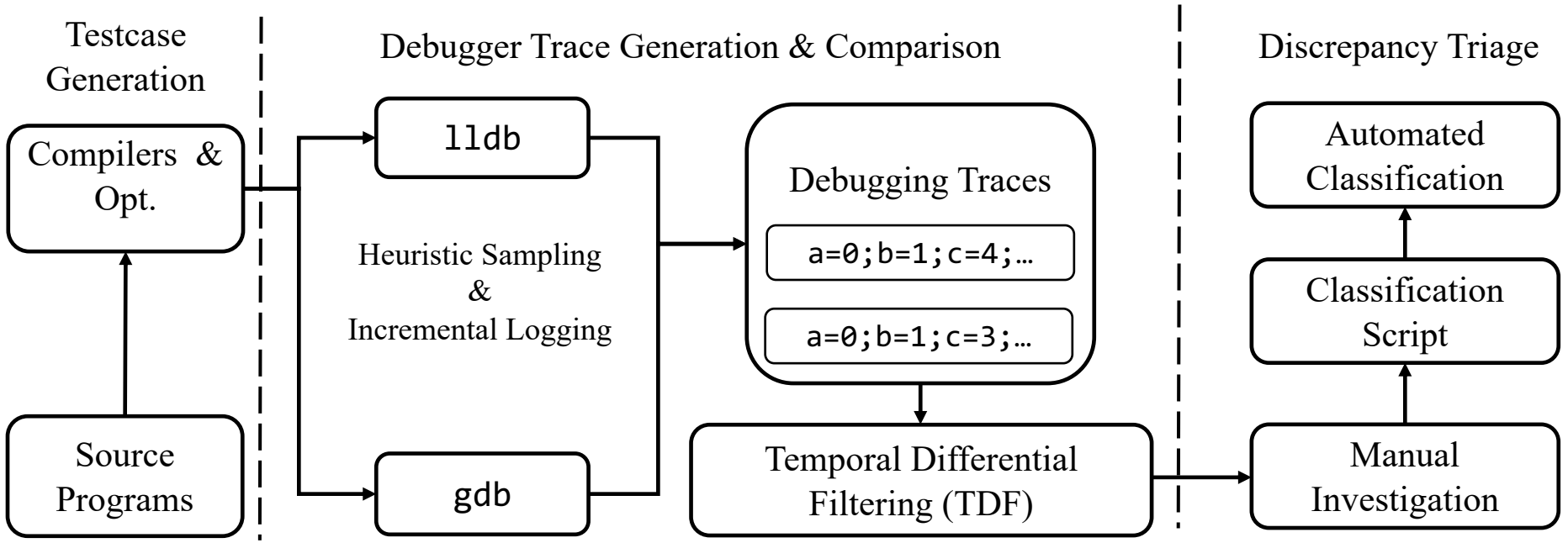
- Collect full **P**rogram states, but **skip lines (E)**.
- or
- **E**xecute every instruction, but **neglect variables (P)**.
- or
- Focus on compilers and optimizations (how they affect DWARF).

	Focus	Tech. category	Target language	Comprehensive	Scalable
DBDB [Lehmann and Pradel 2018]	Debuggers	DT	JavaScript	**P**	?
Tolksdorf et al. [2019]	Debuggers	MT	JavaScript	**P**	?
Debug^2 [Di Luna et al. 2021]	Compiler/Opt.	DT	C/Rust	**E**	✓
Assaiante et al. [2023]	Compiler/Opt.	Static inference	C	**N.A.**	✓
Li et al. [2020]	Compiler/Opt.	DT	C/Rust	**N.A.**	✓
DTD	Debuggers	DT	C	**P,D,WE**	✓

Method

Idea

Same binaries ➡ GDB/LLDB ➡ Same results

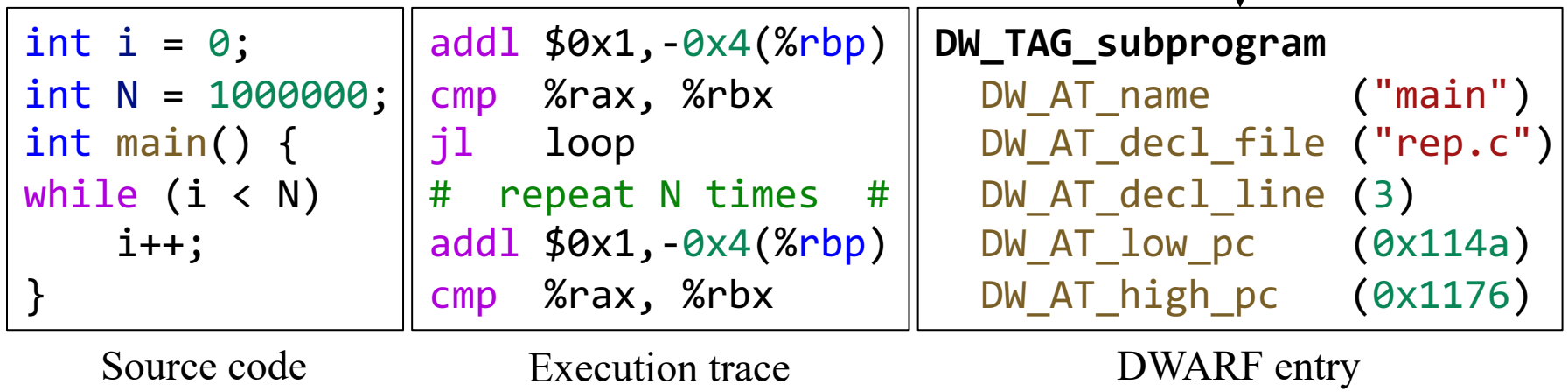


Challenge 1: Inefficiency

Debugger waste time on **loops**.

1 million steps, but only 1 entry.

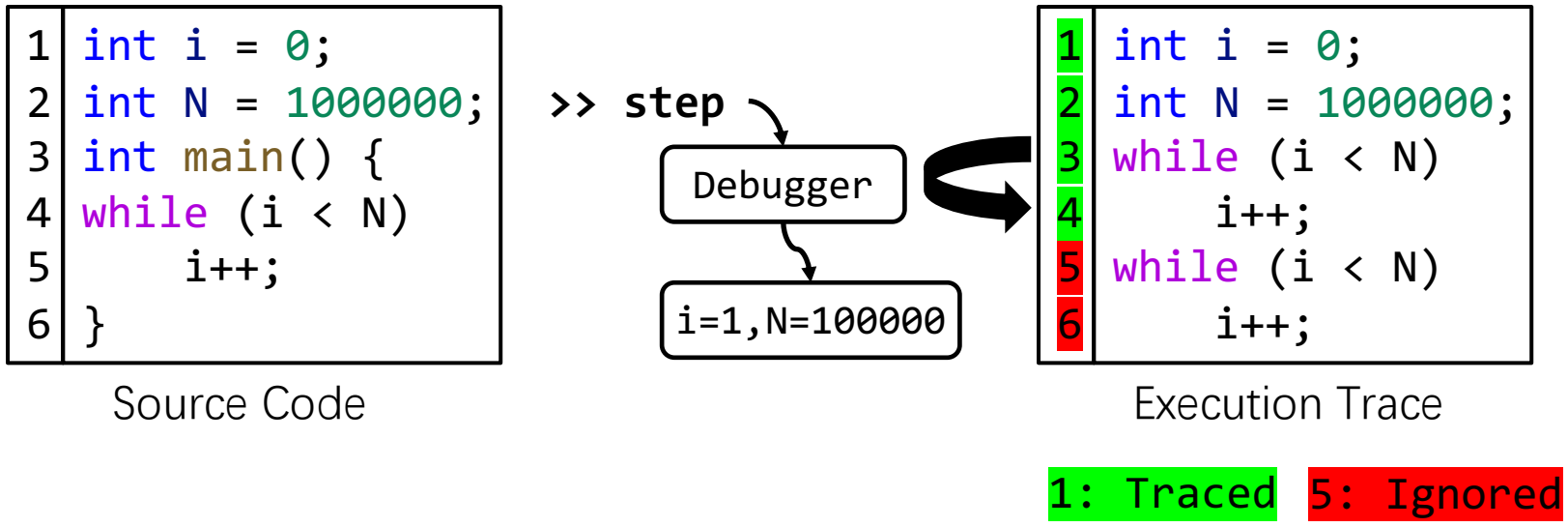
```
>> step * 1000000
```



Solution: PC-guided Heuristics

DWARF maps PC to debug info.

So, we only look at **unique PC's**, but still can cover **every** DWARF entry.



Optimization: Incremental Logging

Arrays **bloat** collected data-facts.

We only track **changed** part.

```
1  int f() {
2      int a[100][100];
3      a[0][0] = 1;
4      a[70][32] = 10;
5  }
```

C source codes

Incremental Log	Bytes
a:{{…},…,{…}}	40000
a:{0:{0:1}}	24
a:{70:{32:10}}	24
...	…

Incremental log

Challenge 2: Uninitialized Variable

Uninitialized variable has **random** value.

Causes **false positive** when comparing LLDB/GDB.

```
1  void f() {
2      int i, uninit[5];
3      uninit[0] = 8;
4      uninit[2] = 5;
5      for (i = 0; i < 5; i++)
6          uninit[i] = i;
7  }
```

C source codes

0	1	2	3	4
?	?	?	?	?
8	?	?	?	?
8	?	5	?	?
8	?	5	?	?
1	2	3	4	5
1	2	3	4	5

Elements of `uninit`

`print uninit[0]`

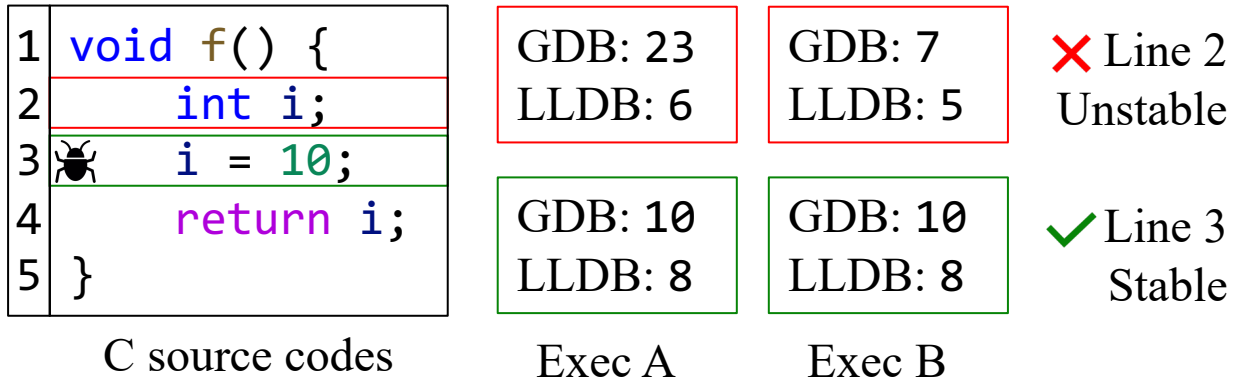
LLDB: `uninit[0] = 10`

GDB: `uninit[0] = 1`

False Positive!

Solution: Temporal Differential Filtering

For each discrepancy, we execute it **twice** and only keep **stable results**.



Evaluation

Evaluation Setup

- Two tested debugger: GDB/LLDB
- Two tested compilers: GCC/Clang
- Four optimization levels: `{-O0,-O1,-O2,-O3}`

Total #programs generated by Csmith	10,000
Total #programs w.o. undefined behavior	7,044
Total #executables used to test debuggers	56,352
Total LOC in Csmith generated C programs	1.7 M
Total #states in DTD-analyzed traces	15 B
Total #data facts on DTD-analyzed traces	432 M

Findings

33,134 (58%) programs that behave differently in LLDB/GDB.
5,013 of them relate to **variables**;
28,121 of them relate to **line number**.

Compiler	Opt. Level	GDB	LLDB	GDB/LLDB	
		Var. Lost	Var. Lost	Var. Diff.	Ctrl. Flow
Clang	O0	0	0	151	12
	O1	70	13	251	4858
	O2	72	0	209	4518
	O3	90	0	195	2399
GCC	O0	3	0	135	0
	O1	12	2332	216	6728
	O2	4	854	116	5364
	O3	4	161	125	4242

Findings

We locate a total of **18 bugs**; **13 confirmed**; **5 fixed**.

	Group	Compiler	Opt. Level	#	Characteristics	Status
Bug$_1$	G1	Clang	O1,O2,O3	202	Clang emits incomplete DWARF information.	Confirmed
Bug$_2$	G1	Clang	O1,O2,O3	27	Clang emits DWARF information of incompatible types.	Confirmed
Bug$_3$	G1	Clang	O1	1	GDB lacks the support for multi-precision arithmetic.	Fixed
Bug$_4$	G1+G5	GCC	O0,O1,O2,O3	23	GDB selects out-of-scope (future scope) variable values.	Fixed
Bug$_5$	G2+G5	GCC	O1,O2,O3	203*	GDB selects wrong variable value at function entrypoint.	Confirmed
Bug$_6$	G3	Clang, GCC	O1,O2,O3	1,619*	LLDB does not show all in-scope variables in `frame variable`.	Reported
Bug$_7$	G3	Clang, GCC	O1,O2,O3	892	LLDB does not handle `DW_OP_bit_piece` correctly.	Comfirmed
Bug$_8$	G3	GCC	O1,O2,O3	1,898	LLDB fails to evaluate `DW_OP_entry_value`.	Reported
Bug$_9$	G3	GCC	O1,O2,O3	18	LLDB shows `<empty constant data>` for a valid entry.	Reported
Bug$_{10}$	G3	GCC	O1,O2,O3	3,080	LLDB lacks the support of `DW_OP_implicit_pointer`.	Reported
Bug$_{11}$	G3	GCC	O1,O2,O3	1,619*	LLDB does not correctly ignore “empty pc ranges”.	Confirmed
Bug$_{12}$	G5	Clang, GCC	O0,O1,O2,O3	1,208*	LLDB underflows when evaluating bitfields.	Confirmed
Bug$_{13}$	G5	Clang, GCC	O1,O2,O3	203*	LLDB treats `DW_OP_div` as unsigned.	Fixed
Bug$_{14}$	G5	Clang, GCC	O1,O2,O3	203*	LLDB treats `DW_OP_deref_size` as unsigned.	Confirmed
Bug$_{15}$	G5	Clang, GCC	O0,O1,O2,O3	1,208*	LLDB displays `0` for optimized-out variables.	Confirmed
Bug$_{16}$	G5	Clang	O1,O2,O3	203*	Clang emits wrong DWARF information.	Confirmed
Bug$_{17}$	G5	Clang	O1,O2,O3	241	GDB shows `<synthetic pointer>` for non-pointer values.	Fixed
Bug$_{18}$	G2+G6	Clang	O1,O2,O3	4?	GDB shows inconsistent source code and stack frame.	Reported

Case Study: LLDB

```
1 static volatile uint64_t g=0;
2 static const int f() {
3   unsigned int i;
4   for(i=0; (i!=10); i++)
5     ++g;
6 }
7 int main() { f(); }
```

Simplified DWARF of `i`

```
1 DW_OP_constu 4294967295
2 DW_OP_reg    $rax
3 DW_OP_and
4 DW_OP_constu 10
5 DW_OP_minus
6 DW_OP_consts -1
7 DW_OP_div
```

✓ Signed DW_OP_div

```
>> i = ($rax – 10) / (-1)

$rax: 10 -> 0
i: 0 -> 10
```

✗ Unsigned DW_OP_div

```
>> i = ($rax – 10) / 127

$rax: 10 -> 0
i: 0 -> 0
```

Takeaways

- Debuggers are **not as reliable as we thought**.
- Debug info requires more "**domain-specific**" coverage and heuristics.
- Debugger **testcase generation**?

My Wife (Yuqi Qian) & Me (Hongyi Lu)

Thank You!

My Website

Email Me

Project Repo

Case Study: GDB

```
1 Range 0x461d-0x4628:
2 DW_OP_breg0 0 [$rax]
3 DW_OP_convert<DW_ATE_unsigned_72 [0x27]>
4 DW_OP_stack_value
5 [4-byte piece]
```

Unsigned integer of
72 bits > 64 bits

Bug-triggering DWARF

```
GDB: That operation is not available
on integers of more than 8 bytes.
```