高精度数值缺陷分析

报告人：姚培森
浙江大学计算机学院-网安学院

报告日期：2022.11.27

Importance of Software Reliability

PayPal accidentally credits man $92 quadrillion

By Sho Wills, CNN

Updated 1355 GMT (2155 HKT) July 17, 2013

'Gangnam Style' breaks YouTube

By Brandon Griggs, CNN

Updated 1850 GMT (0250 HKT) December 3, 2014

A new software glitch was discovered on Boeing's 737 Max

By Chris Isidore, CNN Business

Updated 1845 GMT (0245 HKT) February 6, 2020

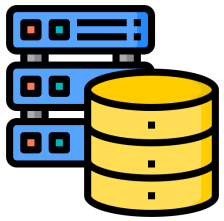
Program Analysis for Software Reliability

Static Analysis

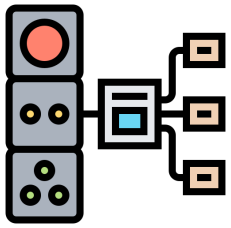
Dynamic Analysis

My Research on Program Analysis

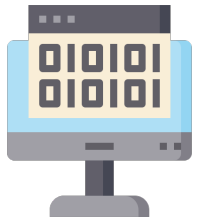
Fundamentals



CFL-reachability
[OOPSLA'22a]



Pointer/Heap Analysis
[PLDI21'a, TOSEM]

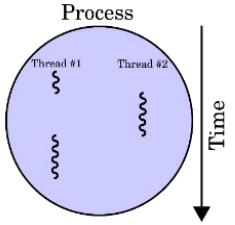


Abstract Interpretation
[OOPSLA'21]

Sparse Analysis
[PLDI'21b]

Applications

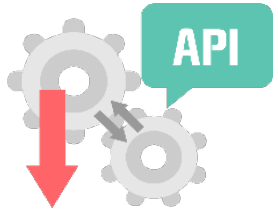
Memory Safety
[S&P'20; S&P'22]



Concurrency Bugs
[PLDI'21a; SEC'23]

Logical Bugs
[ESEC/FSE'21; ISSTA'21]

Numerical Bugs
[ICSE'22]



API Misuses
[ASE'21]

Performance Bugs
[OOPSLA'22b]

Research Impact

- Found 1500+ bugs (~**80** CVE) in open-source software

Outline

- Introduction: Divide-by-Zero Bug Finding
- Improve Precision via Affirmative Evidence
- Improve Precision via Exists-Forall Solving
- Conclusion and Future Work

Divide-by-Zero Bugs


```
int division(int x, int y){
    return (x / y);
}
```

🤔 What if the divisor is zero?

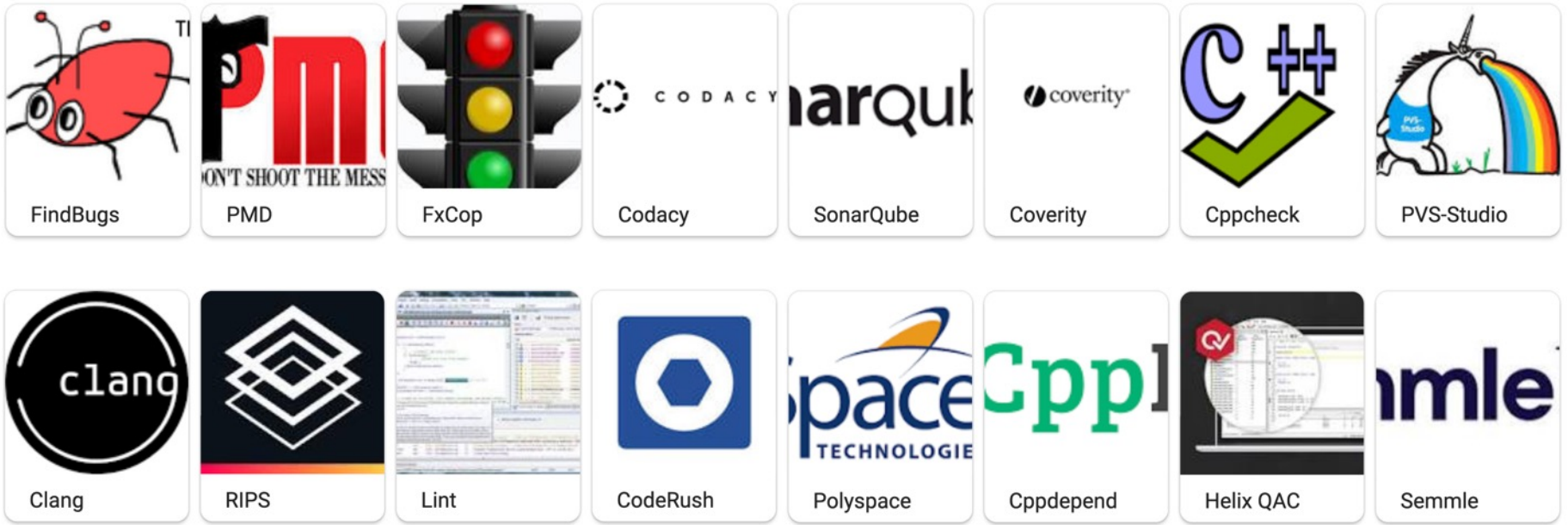
How to find the divide-by-zero bug?

- Dynamic approaches
  Fuzzing, AddressSanitizer, …
- Static approaches
  Bug finding, formal verification

We focus on ***precise static bug finding*** for **large, real-world programs**

What is Static Bug Finding?

Finding bugs in software without executing it



Who are Using Static Bug Finding?

Almost all software companies we know

SYNOPSYS®

HUAWEI

Microsoft

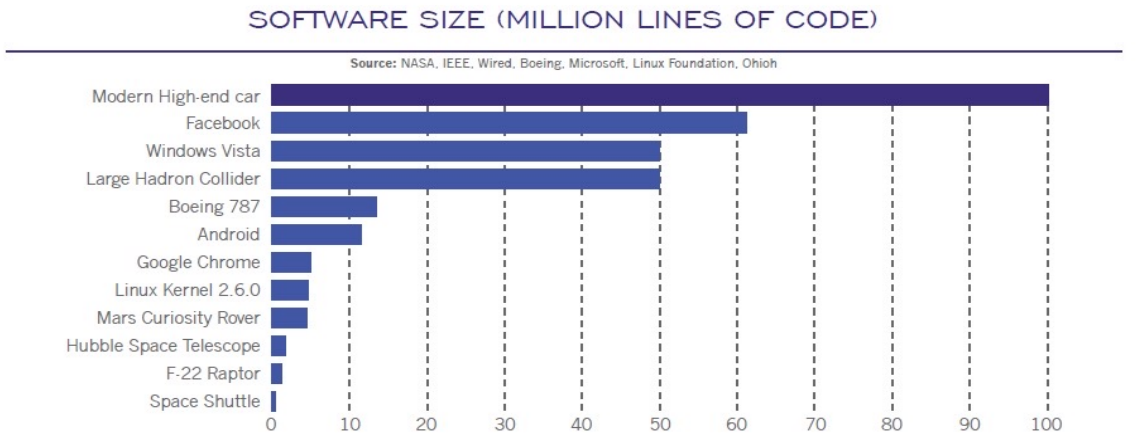
Baidu 百度

Alibaba

”It prevents hundreds of bugs per day from entering the Google codebase.” ---- *CACM 2018*

Finding Divide-by-Zero Bugs Precisely?

- Industry-strength static analyzers: >70% false positive rates when handling large-scale programs
  - Clang Static Analyzer (CSA), Infer,…



Why is the unsatisfying precision?

- Even {flow-, context-, …} sensitive analyses…

Outline

- Introduction: Divide-by-Zero Bug Finding
- **Improve Precision via Affirmative Evidence [1]**
- Improve Precision via Exists-Forall Solving
- Conclusion and Future Work

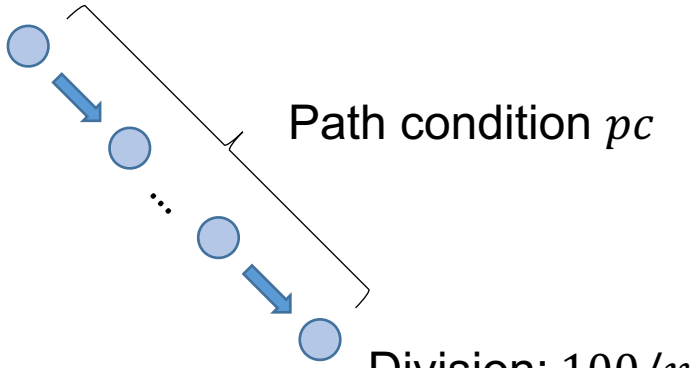
[1] Precise Divide-By-Zero Detection with Affirmative Evidence. ICSE'22
Yiyuan Guo, Jinguo Zhou, Peisen Yao, Qingkai Shi, and Charles Zhang.

Conventional Paradigm to Finding Bugs

The analyzer reports a divide-by-zero bug if:

The divisor variable v ***may*** be zero upon a path p

i.e., the constraint $\boldsymbol{pc \wedge v = 0}$ is satisfiable



Execution path p

When analyzing real-world programs:

v = lib();	**unmodelled libraries**
asm ("mov %1, %0" : “=r” (v) ..);	**unmodelled semantics**
v = nondet();	**analysis’s approximations**

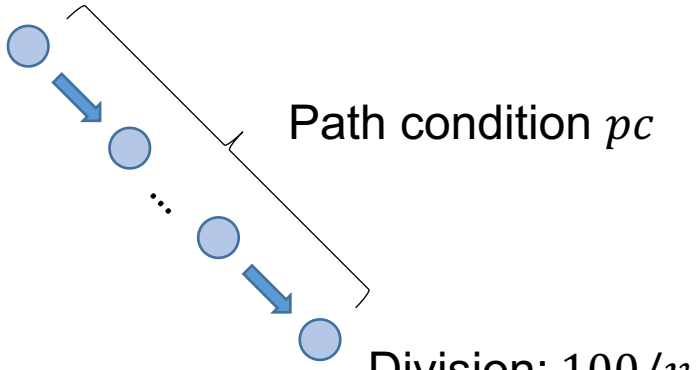
Under-constrained variables in static analysis!

Precision Loss due to Under-constrained Variables

The analyzer reports a divide-by-zero bug if:

The divisor variable v ***may*** be zero upon a path p

i.e., the constraint $\boldsymbol{pc \wedge v = 0}$ is satisfiable



Under-constrained variables can easily *make the query satisfiable*!

Massive false positives!

Our Key Insight

- **Conventional paradigm**: report a bug when safety verification fails
  - May not prove that $pc \wedge v = 0$ is unsatisfiable
  - Susceptible to under-constrained variables

- **Our work**: actively find affirmative evidence for triggering the bug
  - Serves as extra information on the under-constrained variables
  - Helps improving the precision of the analysis

Motivating Example


```
1 void foo(int dx, int dy, int d) {
2     int diff;
3     if (dx >= dy && dy >= d)
4         diff = 1 + dx - dy;
5     else
6         diff = -1;
7
8     print(100 / diff);
9
10    int dz = 2*dx - (dy+d);
11
12    print(100 / dx);
13    print(100 / dy);
14    print(100 / dz);
15 }
```

- Suppose the caller of the function `foo` is unknown to the static analyzer
- Then the variables `dx`, `dy`, and `d` are under-constrained
- Out of the four divisions, which ones should be reported as potential divide-by-zeros?

Motivating Example


```
1 void foo(int dx, int dy, int d) {
2     int diff;
3     if (dx >= dy && dy >= d)
4         diff = 1 + dx - dy;
5     else
6         diff = -1;
7
8     print(100 / diff);
9
10    int dz = 2*dx - (dy+d);
11
12    print(100 / dx);
13    print(100 / dy);
14    print(100 / dz);
15 }
```

Safe: `diff` is either -1 or no less than 1 ✓

Conventional methods: Lines 12-14 have bugs

Since the variables `dx`, `dy`, and `d` are under-constrained, the confidence of the reports are low

Possibly buggy: `dx`, `dy`, and `dz` may equal zero under all paths ✗

Motivating Example


```
1 void foo(int dx, int dy, int d) {
2     int diff;
3     if (dx >= dy && dy >= d)
4         diff = 1 + dx - dy;
5     else
6         diff = -1;
7
8     print(100 / diff);
9
10    int dz = 2*dx - (dy+d);
11
12    print(100 / dx);
13    print(100 / dy);
14    print(100 / dz);
15 }
```

Can we find evidence to show that some bugs are more plausible?

① The programmer compares `dx` with `dy` and `dy` with `d`

② This suggests her beliefs that `dx=dy` and `dy=d` may hold.

③ If such beliefs hold, `dz = 2*dx - (dy+d) = 0`

It seems that Line 14 is more likely to trigger divide-by-zero

Our approach finds evidence ① - ③ and only reports the bug at Line 14

Problem Statement: Evidence-based Reasoning

In this work, we identify two categories of evidence:

1. Source evidence: an explicit source of "bad" value assigned

```
v = 0
v = atoi(argv[1])
```

Direct assignment of zero

Tainted input

2. Bound evidence: likely facts produced by belief analysis

```
if (v1 >= v2) ...
```

Bound checking statement

```
v1 = v2
```

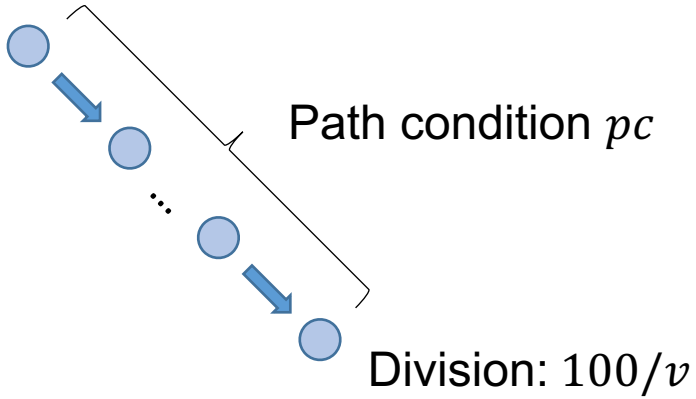
Fact that is likely to hold

Will focus on bound evidence in this talk

Problem Statement: Evidence-based Reasoning

Our bug detection criteria:

1. Constraints for divide-by-zero is satisfiable: SAT($pc \wedge v = 0$)
2. The discovered **bound evidences can enforce v to be zero**



Execution path p

If the evidences hold, v **must be zero**

Approach: Path-sensitive Evidence Propagation


```
1 void foo(int dx, int dy, int d) {
2     int diff;
3     if (dx >= dy && dy >= d)
4         diff = 1 + dx - dy;
5     else
6         diff = -1;
7
8     print(100 / diff);
9
10    int dz = 2*dx - (dy+d);
11
12    print(100 / dx);
13    print(100 / dy);
14    print(100 / dz);
15 }
```

- *Path-sensitivity* is needed to prove `diff` non-zero
- *Evidence propagation* is needed to infer `dz=0` from the bound evidence `dx = dy` and `dy = d`

Goal: Simultaneously achieve these two

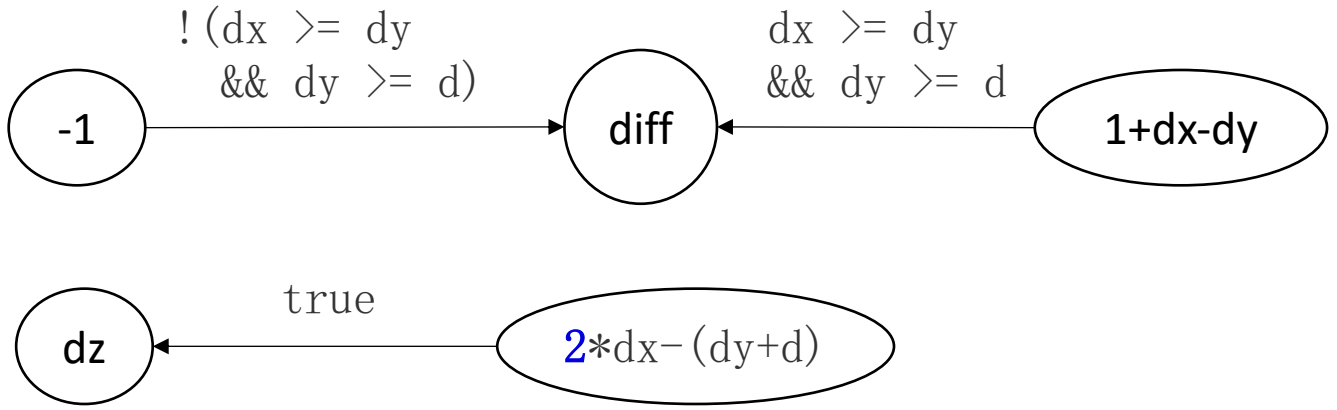
Path-sensitive Evidence Propagation

- Efficient path-sensitive analysis:
  - Data dependence graph for skipping irrelevant paths [Shi et al., PLDI'18]

- Evidence propagation:
  - Source evidence: a taint analysis to mark the tainted variables
  - Bound evidence: a dedicated symbolic domain Γ
    - Encode bound evidence as constraints on Γ
    - Constraints are enforced during iterative propagation

Path-sensitive Evidence Propagation

```
1 void foo(int dx, int dy, int d) {
2     int diff;
3     if (dx >= dy && dy >= d)
4         diff = 1 + dx - dy;
5     else
6         diff = -1;
7
8     print(100 / diff);
9
10    int dz = 2*dx - (dy+d);
11
12    print(100 / dx);
13    print(100 / dy);
14    print(100 / dz);
15 }
```



***Augmented data dependence graph* G**

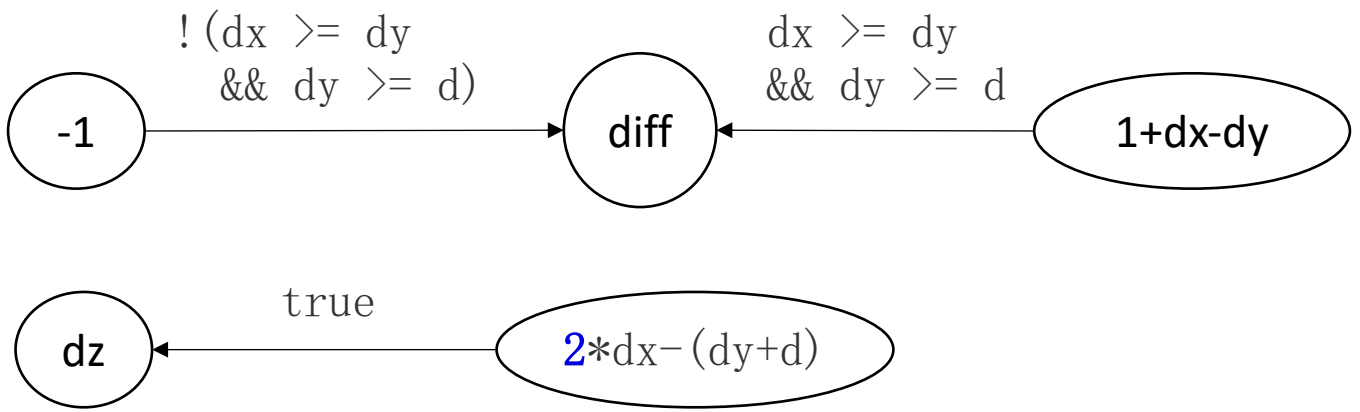
Initial constraints for Γ extracted from G :

$$\Gamma(\text{diff}) = \{(-1, \neg\text{cond}), (1 + \text{dx} - \text{dy}, \text{cond})\},$$
$$\text{cond} = \text{dx} \geq dy \wedge dy \geq d$$

$$\Gamma(\text{dz}) = \{(2 \times dx - (dy + d), true)\}$$

Bound evidence: dx = dy, dy = d

Path-sensitive Evidence Propagation



Initial constraints for Γ extracted from G :

① $\Gamma(\text{diff}) = \{(-1, \neg\text{cond}), (1 + \text{dx} - \text{dy}, \text{cond})\},$
$\text{cond} = \text{dx} \geq dy \wedge dy \geq d$

$\Gamma(\text{dz}) = \{(2{\times}dx\ - (dy + d), true)\}$

Extra constraints based on evidence:

② $\Gamma(\text{dx}) = \Gamma(\text{d}y)$
$\Gamma(\text{d}y) = \Gamma(\text{d})$

Resolving constraints ① ② : $\Rightarrow\ 0 \in \Gamma(dz)$

Divide-by-zero bug for the variable dz

Evaluation: Implementation and Setup

- ***WIT***: powered by Pinpoint@Ant Group

- **Subjects**: 12 open-source projects
  - Popularity: > 10k stars in GitHub
  - Generality: different sizes and functionalities
  - At least 3.3 divisions per 1K Loc

Table 2: Selected projects for evaluation.

Project	Loc	#Div/KLoC
masscan	34k	5.4
goaccess	53k	1.1
libuv	59k	0.8
redis	131k	5.0
git	226k	4.5
vim	354k	1.8
ImageMagick	382k	6.6
openssl	465k	4.1
systemd	600k	5.0
php	1,012k	1.3
gdb	1,932k	1.6
Linux kernel	15,164k	2.1

Evaluation: Research Questions

- **RQ1**: The effectiveness of *WIT*
  - **RQ1.1**: Precision improvement by evidence
  - **RQ1.2**: Bug detection capability
- **RQ2**: Comparison with existing analyzers
  - Clang Static Analyzer (CSA) https://clang-analyzer.llvm.org/
  - Facebook Infer https://github.com/facebook/infer

Table 2: Selected projects for evaluation.

Project	Loc	#Div/KLoC
masscan	34k	5.4
goaccess	53k	1.1
libuv	59k	0.8
redis	131k	5.0
git	226k	4.5
vim	354k	1.8
ImageMagick	382k	6.6
openssl	465k	4.1
systemd	600k	5.0
php	1,012k	1.3
gdb	1,932k	1.6
Linux kernel	15,164k	2.1

RQ1.1: Precision Improvement by Evidence

- Comparing with an evidence agnostic variant WIT^-
  - Report divide-by-zero **without** considering evidence

Table 3: Divide-by-zero detection on real-world projects. WIT$^-$ represents a variant of WIT unaware of evidence.

Project	# of reports		FP rate		Analysis time	
	WIT	WIT$^-$	WIT	WIT$^-$	WIT	WIT$^-$
masscan	3	10	30%	80%	4m24s	6m
goaccess	2	15	0	80%	1m38s	1m33s
libuv	1	3	0	67%	1m32s	1m29s
redis	1	20	0	95%	23m12s	23m9s
git	10	29	40%	79%	38m3s	34m34s
vim	4	32	25%	88%	109m55s	109m12s
ImageMagick	5	47	20%	89%	196m39s	189m10s
openssl	2	18	0	89%	31m55s	31m48s
systemd	5	20	60%	90%	202m8s	183m21s
php	4	17	50%	88%	104m38s	94m21s
gdb	5	213	20%	96%	248m28s	299m1s
Linux kernel	53	2839	19%	NA	425m35s	452m12s

The precision of WIT (22%) greatly outperforms WIT^- (86%)

RQ1.2: Bug Detection Capability

Missed bugs compared to WIT^-

Table 4: Distribution of true positives reported by Wit (the column "Total") into Class Src and Class Bd. The column "Missed" shows the number of true positives reported by Wit$^-$ but missed by Wit.

Project	Total	Src	Bd	Missed
masscan	2	2	0	0
goaccess	2	2	0	1
libuv	1	0	1	0
redis	1	1	0	0
git	6	4	2	0
vim	3	1	2	1
ImageMagick	4	3	2	1
openssl	2	1	1	0
systemd	2	1	1	0
php	2	0	2	0
gdb	4	3	1	4
Linux kernel	43	36	10	NA

1. Adding the criteria of evidence only **misses a small portion** (**12%**) of bugs detected by WIT^-

Find real bugs with *WIT*

Table 5: Divide-by-zero bugs confirmed by developers.

git	Linux	gdb	Image Magick	goaccess	libuv	openssl	vim	systemd
1	4	1	2	2	1	1	1	1

```
From: Yiyuan GUO @ 2021-05-14  9:16 UTC (permalink / raw)
  To: hare; +Cc: axboe, linux-block, yguoaz, Yiyuan GUO

In function blk_mq_map_queues, qmap->nr_queues may equal zero
and thus it needs to be checked before we pass it to function
queue_index.

Signed-off-by: Yiyuan GUO <yguoaz@cse.ust.hk>
---
 block/blk-mq-cpumap.c | 3 ++-
 1 file changed, 2 insertions(+), 1 deletion(-)

diff --git a/block/blk-mq-cpumap.c b/block/blk-mq-cpumap.c
index 3db84d319..dc440870e 100644
--- a/block/blk-mq-cpumap.c
+++ b/block/blk-mq-cpumap.c
@@ -65,7 +65,8 @@ int blk_mq_map_queues(struct blk_mq_queue_map *qmap)
                } else {
                        first_sibling = get_first_sibling(cpu);
                        if (first_sibling == cpu)
-                               map[cpu] = queue_index(qmap, nr_queues, q++);
+                               if (nr_queues)
+                                       map[cpu] = queue_index(qmap, nr_queues, q++);
                        else
                                map[cpu] = map[first_sibling];
                }
--
2.25.1
```

2. *WIT* has detected **14** divide-by-zero bugs confirmed by the developers

RQ2: Comparison with Existing Analyzers

- Comparing with Clang Static Analyzer(CSA) and Infer

Table 7: Divide-by-zero detection results for Infer and CSA. NA denotes the false positive rate when no bug is reported.

Project	# of reports		FP rate		Analysis time	
	Infer	CSA	Infer	CSA	Infer	CSA
masscan	0	0	NA	NA	1m4s	4m8s
goaccess	1	0	0	NA	3m27s	10m59s
libuv	0	0	NA	NA	3m13s	4m47s
redis	4	1	50%	0	8m47s	15m37s
git	7	0	100%	NA	8m55s	15min5s
vim	0	2	NA	50%	24m21s	17m47s
ImageMagick	7	2	71%	100%	16m7s	25m24s
openssl	0	1	NA	0	49m2s	9m43s
systemd	1	0	100%	NA	31m21s	11m45s
php	0	2	NA	100%	20m23s	57m28s
gdb	0	6	NA	83%	41m59s	81m17s
Linux kernel	Crash	63	Crash	51%	Crash	281m4s

1. Infer and CSA have relative low recalls and missed many bugs found by *WIT*

2. *WIT* also has lower false positive rate compared to Infer (64%) and CSA (55%)

WIT is significantly more precise and sometimes even detects more divide-by-zero bugs

Summary

- Precision issues in static detection of divide-by-zero
- Under-constrained variables causing the imprecision
- Path-sensitive affirmative evidence propagation to find divide-by-zero bugs with high confidence

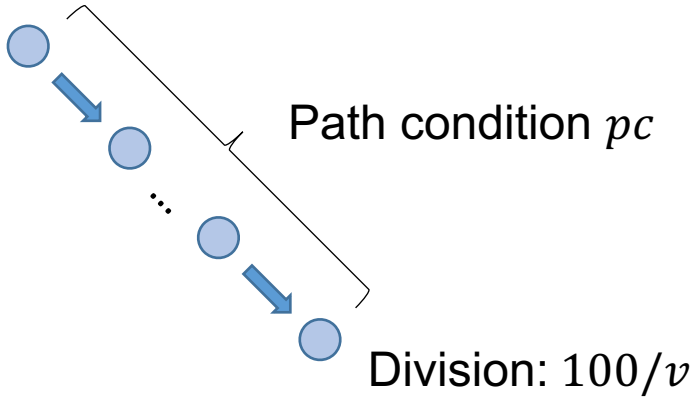
Outline

- Introduction: Divide-by-Zero Bug Finding
- Improve Precision via Affirmative Evidence
- Improve Precision via Exists-Forall Solving
- Conclusion and Future Work

Recap: The Evidence-based Numerical Analysis

Our bug detection criteria:

1. Constraints for divide-by-zero is satisfiable: SAT($pc \wedge v = 0$)
2. A set of discovered **evidences that can enforce v to be zero**



What if we do not have the evidences? ☹

Motivating Example II

```
1  int main () {
2    int a, b = input();
3    int x = lib1(a);
4    int y = lib2(a,b);
5    if (a * (x - y) + b > 0) {
6        print(x / 0);
7    }
8    …
9  }
```

- lib1 and lib2: unknown library functions
- The variables `x` and `y` can be regarded as **under-constrained variables**
- Report a divide-by-zero bug for Line 6 ?

Path condition `a*(x - y)+b>0` is satisfiable, but …

Motivating Example II

```
1  int main () {
2    int a, b = input();
3    int x = lib1(a);
4    int y = lib2(a,b);
5    if (a * (x - y) + b > 0) {
6        print(x / 0);
7    }
8    …
9  }
```

- lib1 and lib2: unknown library functions
- The variables `x` and `y` can be regarded as **under-constrained variables**
- Report a divide-by-zero bug for Line 6 ?

Consider the path condition $a*(x-y)+b>0$

- ***Just report to developers?***: might be difficult to confirm
- ***Generate concrete inputs?***: may violate the semantics of lib1, lib2

Our Key Insight II

- **Conventional paradigm**: report a bug whenever safety verification fails,
  - Prove whether $pc \wedge v = 0$ is unsatisfiable
  - Susceptible to under-constrained variables.

- **No affirmative evidence** on the under-constrained variables

- **Our second work**: use exists-forall solving to **eliminate the effects** of the under-constrained variables

$$\exists P.\forall Q\ .\ \varphi(P, Q)$$

where Q is the set of under-constrained variables

Insight by Example

```
1  int main () {
2    int a, b = input();
3    int x = lib1(a);
4    int y = lib2(a,b);
5    if (a * (x - y) + b > 0) {
6        print(x / 0);
7    }
8    …
9  }
```

The variables `x` and `y` can be regarded as under-constrained variables

Report a divide-by-zero bug for Line 6 ?

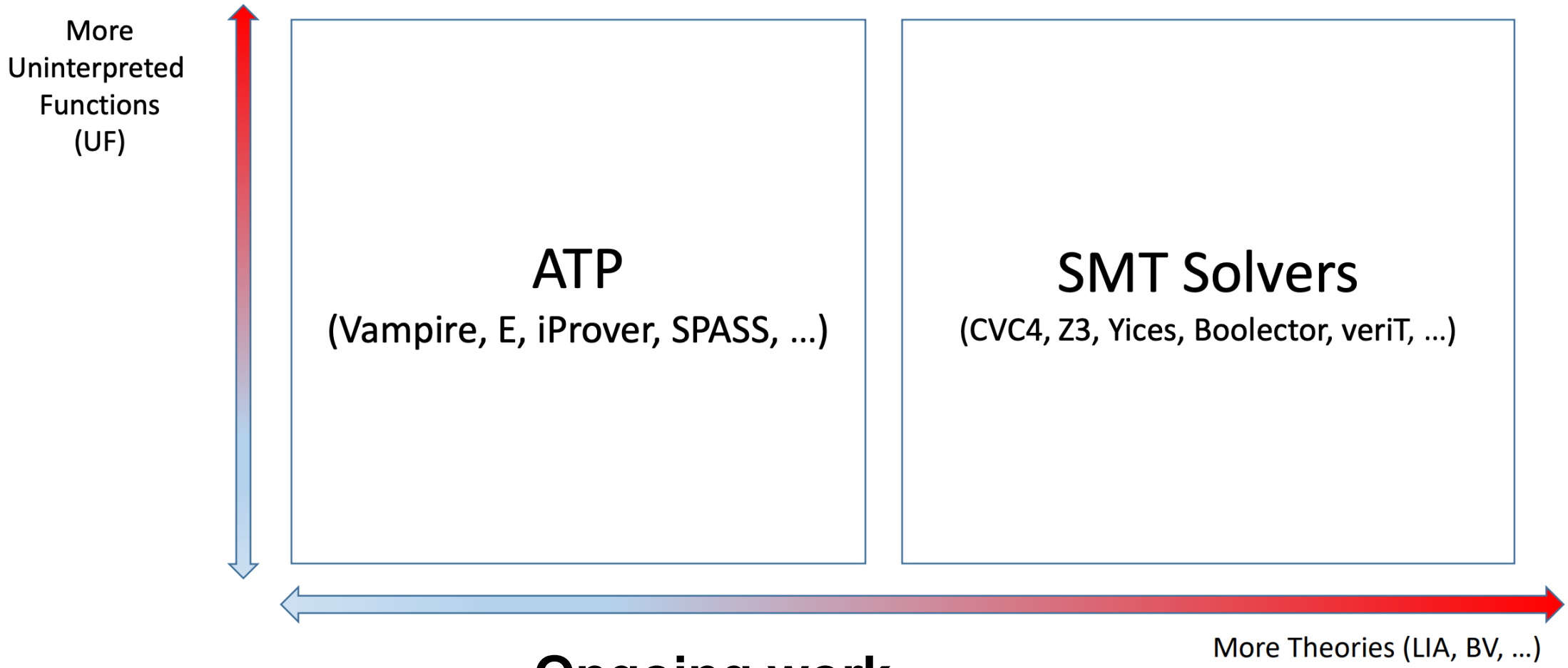
Solving the condition $a*(x-y)+b>0$?

Solving $\exists a, b. \forall x, y \; . \; a * (x - y) + b > 0$!

A "good enough" assignment is **{a = 0, b = 1}**

Challenge: Efficient Quantifier Reasoning

How to efficiently solve $\exists P.\forall Q\ .\ \varphi(P,Q)$**?**



Ongoing work…

Outline

- Introduction: Divide-by-Zero Bug Finding
- Improve Precision via Affirmative Evidence
- Improve Precision via Exist-Forall Solving
- Conclusion and Future Work

Conclusion

- Under-constrained variables can cause imprecision

- Two strategies for improving the precision
  1. **Find additional "facts"** of the variables via **affirmative evidence**
  2. **Eliminate the effects** of the variables via **exists-forall solving**

- Find numerical bugs with high confidence
  - May also be easier for the developers to confirm

Static Bug Finding with Zero False Positive?

- A recent trend in static bug finding
  - ***Sound(y), but high FP rates → "Actively unsound" for low FP rates***

Group	Paper	Conference
Infer	Incorrectness logic	POPL'19
BinSec	Not all bugs are created equal, but robust reachability can tell the difference	CAV'21
Pinpoint	Precise divide-by-zero detection with affirmative evidence [1st part of this talk]	ICSE'22

- Limitations of existing efforts
  - Capability: theoretical framework vs. bug-specific technique?
  - Performance: may introduce additional overhead
  - Coverage: how many bugs are missed?

感谢观看

rainoftime.github.io