Infer: A Static Program Analyser

Sangharatna Godboley

School of Computing, NUS Singapore

February 6, 2020

Seminar Outline

Astree (sound)

- Proves the absence of runtime errors and undefined behaviour in C programs
- Used to prove absence of runtime errors in Airbus flight control software and Docking software for the International Space Station
- Many man-years of effort (since 2001) to develop
- See www.astree.ens.fr/

Coverity (neither sound nor complete)

- Looks for bugs in C, C++, Java, and C#.
- Used by: a. >1100 companies. b. NASA JPL (in addition to many other tools).
- Offered as a free, cloud-based service for open-source projects.
- See www.coverity.com

Java PathFinder (sound but can be imprecise)

- Finds bugs in mission-critical Java code.
- Developed by NASA.
- Focuses on concurrency errors (race conditions), uncaught exceptions.
- Free and open source!
- See babelfish.arc.nasa.gov/trac/jpf

Clang Static Analyser (neither sound nor complete)

- Part of llvm compiler infrastructure; works only on C and Objective-C programs
- Over 30 checks built into default analyzer
- Built for debugging iOS apps, so includes extensive functionality for finding memory problems
- Can suppress false positives reported by tool by adding annotations to code
- Very snappy IDE Integration with Xcode
- Support for C++ coming soon!
- Free and open source!
- http://clang-analyzer.llvm.org/

Infer Static Program Analyser (neither sound nor complete)

- Infer is a static program analyzer for Java, C, and Objective-C, written in OCaml.
- Infer is deployed within Facebook and it is running continuously to verify select properties of every code modification for the main Facebook apps for Android and iOS, Facebook Messenger, Instagram, and other apps.
- At present Infer is tracking problems caused by null pointer dereferences and resource and memory leaks, which cause some of the more important problems on mobile.
- Infer came to Facebook with the acquisition of the verification startup Monoidics in 2013. Monoidics was itself based on recent academic research, particularly on separation logic and bi-abduction.

Infer Static Program Analyser (neither sound nor complete)

- Infer.AI, a general analysis framework which is an interface to the modular analysis engine which can be used by other kinds of program analyses (technically, called "abstract interpretations", hence the AI monicker).
- This added generality has been used to develop instantiations of Infer.AI for security, concurrency and in other domains.
- Free and open source tool
- https://fbinfer.com/docs/about-Infer.html

Important Terms

	POSITIVE (Bug Found)	NEGATIVE (Bug Not-Found)
TRUE	**TRUE POSITIVE** -> Reports that are correct (i.e. exisiting bugs)	**TRUE NEGATIVE** -> Program is safe and tool confirms it
FALSE	**FALSE POSITIVE** -> Reports that are incorrect (i.e. non-exisiting bugs) where non-exisitant bugs are reported as bugs.	**FALSE NEGATIVE** -> Undiscovered bugs -> Potential bugs missed by tool

Figure 1: Confusion Matrix

Findings Bugs with Compiler Techniques

Compile-time warnings

- clang t.c
  t.c:38:13: warning: invalid conversion '%lb'
  printf("%s%lb%d", "unix", 10, 20);

Static Analysis

- Checking performed by compiler warnings inherently limited
- Find path-specific bugs
- Deeper bugs: memory leaks, buffer overruns, logic errors

Why Analyze Source Code?

Bug-finding requires excellent diagnostics

- Tool must explain a bug to the user
- Users cannot fix bugs they don't understand
- Need rich source and type information

What about analyzing LLVM IR?

- Loss of source information
- High-level types discarded
- Compiler lowers language constructs
- Compiler makes assumptions (e.g., order of evaluation)

Benefits of Static Analysis

Early discovery of bugs

- Find bugs early, while the developer is hacking on their code
- Bugs caught early are cheaper to fix

Systematic checking of all code

- Static analysis reasons about all corner cases

Find bugs without test cases

- Useful for finding bugs in hard-to-test code
- Not a replacement for testing

How does static analysis work?

- Can catch bugs with different degrees of analysis sophistication
- Per-statement, per-function, whole-program all important

How does static analysis work?

```
int f(int y) {
  int x;

  if (y)
    x = 1;

  printf("%d\n", y);

  return x;
}
```

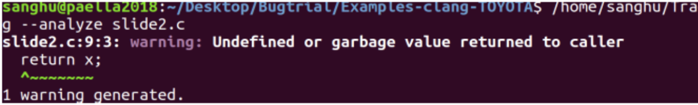


Figure 2: Example1-1

How does static analysis work?

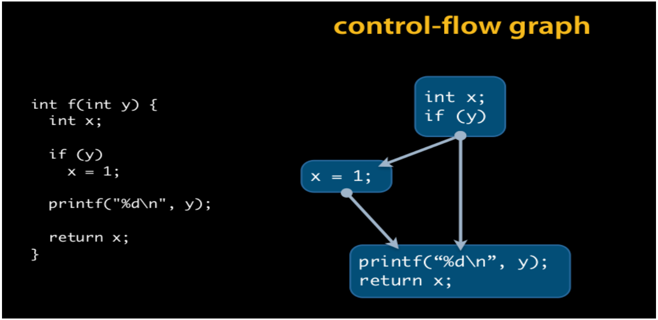


Figure 3: Example1-2

How does static analysis work?

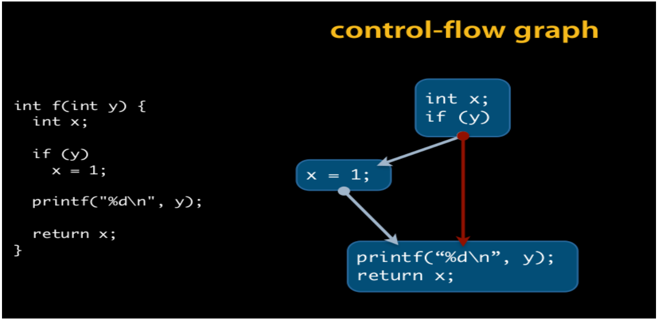


Figure 4: Example1-3

How does static analysis work?

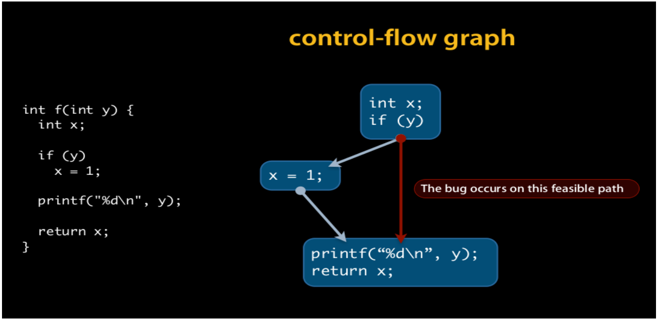


Figure 5: Example1-4

How does static analysis work?

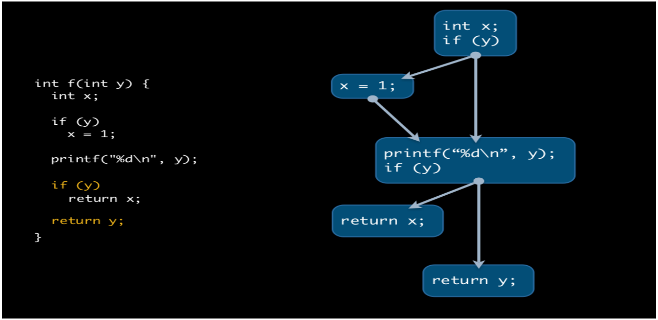


Figure 6: Example2-1

How does static analysis work?

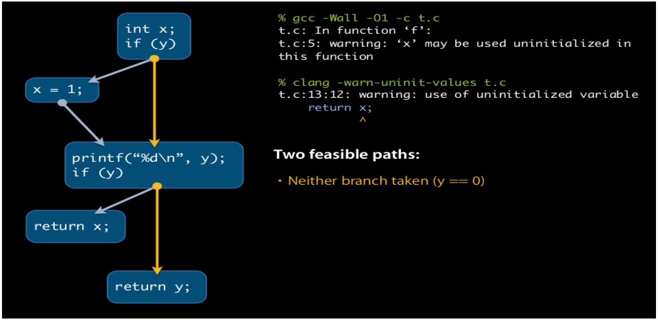


Figure 7: Example2-3

How does static analysis work?

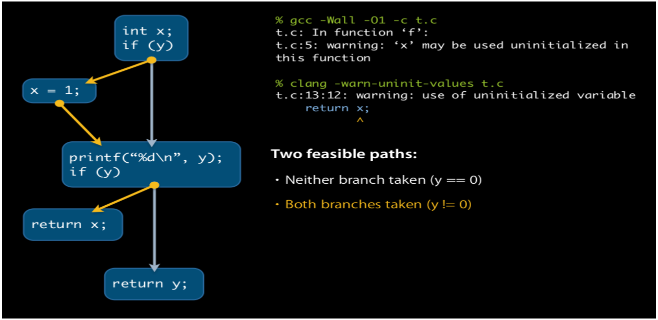


Figure 8: Example2-4

How does static analysis work?

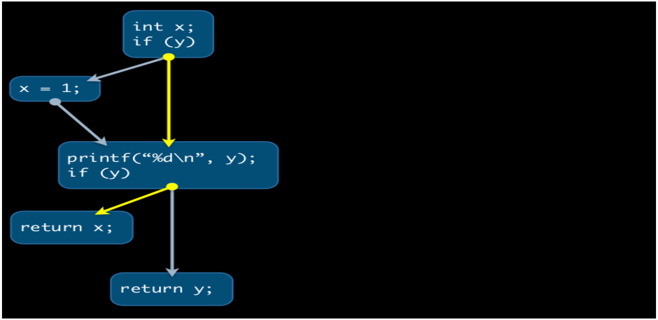


Figure 9: Example2-5

How does static analysis work?

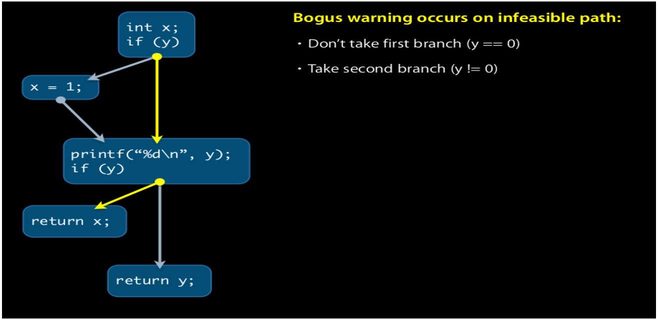


Figure 10: Example2-6

False Positives (Bogus Errors)

False positives can occur due to analysis imprecision

- False paths
- Insufficient knowledge about the program

Many ways to reduce false positives

- More precise analysis
- Difficult to eliminate false positives completely

Flow-Sensitive Analysis

Flow-sensitive analysis reason about flow of values

- $y = 1;$
- $x = y + 2; //x == 3$

No path-specific information

- $if(x == 0)$
- $++x; //x ==?$
- $else$
- $x = 2; //x == 2$
- $y = x; //x ==?, y ==?$

Path-Sensitive Analysis

Reason about individual paths and guards on branches

- $if(x == 0)$
- $++x; //x == 1$
- $else$
- $x = 2; //x == 2$
- $y = x; //(x == 1, y == 1) or (x == 2, y == 2)$

Uninitialized variables example:

- Path-sensitive analysis picks up only 2 paths
- No false positive

Path-Sensitive Analyses

Worst-case exponential-time

- Complexity explodes with branches and loops
- Lots of clever tricks to reduce complexity in practice

"Clang and Infer static analysers use flow and partial path-sensitive analyses"

Infer: Installation

- On Linux, use latest binary release.
- Download the tarball then extract it anywhere on your system to start using infer.
- For example, this downloads infer in /opt on Linux (replace VERSION with the latest release, eg VERSION=0.17.0)

Infer: Installation

```
VERSION=0.XX.Y; \
curl -sSL "https://github.com/facebook/infer/releases/download/v$VERSION/infer-linux64-v$VERSION.tar.xz" \
| sudo tar -C /opt -xJ && \
ln -s "/opt/infer-linux64-v$VERSION/bin/infer" /usr/local/bin/infer
```

Figure 11: Infer Installation on Linux

Infer: Two Phases

The capture phase

- Compilation commands are captured by Infer to translate the files to be analyzed into Infer's own internal intermediate language.
- This translation is similar to compilation, so Infer takes information from the compilation process to perform its own translation.
- This is why infer called with a compilation command: *infer run – clang -c file.c*.
- What happens is that the files get compiled as usual, and they also get translated by Infer to be analyzed in the second phase.
- In particular, if no file gets compiled, also no file will be analyzed.
- Infer stores the intermediate files in the results directory which by default is created in the folder where the infer command is invoked, and is called infer-out/.

Infer: Two Phases

The analysis phase

- In this phase, the files in infer-out/ are analyzed by Infer.
- Infer analyzes each function and method separately. If Infer encounters an error when analyzing a method or function, it stops there for that method or function, but will continue the analysis of other methods and functions.
- So, a possible workflow would be to run Infer on your code, fix the errors generated, and run it again to find possibly more errors or to check that all the errors have been fixed.
- The errors will be displayed in the standard output and also in a file infer-out/bugs.txt.
- Infer filters the bugs and show the ones that are most likely to be real.
- In the results directory (infer-out/), however, it saves a file report.csv that contains all the errors, warnings and infos reported by Infer in csv format.

Infer: Infer Manuals

Here are the man pages for all the infer commands:

- infer
- infer-analyze
- infer-capture
- infer-compile
- infer-explore
- infer-report
- infer-reportdiff
- infer-run[a] (Covers *capture* and *analyze*)

[a] https://fbinfer.com/static/man/infer-run.1.html

Folder Structure

```
infer-out
├── captured/
├── log/
├── specs/
├── bugs.txt
├── report.json
├── toplevel.log
└── ...
```

Figure 12: Files and folders inside the infer-out folder

Infer: Bug Types

- *captured/* contains information for each file analyzed by Infer.
- *specs/* contains the specs of each function that was analyzed, as inferred by Infer.
- *log/* and *toplevel.log* contains logs
- *bugs.txt* and report.json contain the Infer reports in text and JSON formats
- there are other folders reserved for Infer's internal workings

Infer: Bug Types

- Checkers Immutable Cast
- Deadlock
- Dead store
- Empty Vector Access
- Field should be nullable
- Fragment retains view
- Interface not thread-safe
- Ivar not null checked
- Lock Consistency Violation
- Memory leak
- Mixed self weakSelf
- Multiple weakSelf Use
- Null dereference

Infer: Bug Types

- Parameter not null checked
- Premature nil termination argument
- Resource leak
- Retain cycle
- Static initialization order fiasco
- Strict mode violation
- StrongSelf Not Checked
- Thread-safety violation
- UI Thread Starvation
- Unsafe_GuardedBy_Access

Infer: Bug Types

Dead Store

This error is reported in C/C++. It fires when the value assigned to a variables is never used (e.g., int i = 1; i = 2; return i;).

Infer: Bug Types

Memory leak in C

- This error type is only reported in C and Objective-C code. In Java we do not report memory leaks because it is a garbage collected language.
- In C, Infer reports memory leaks when objects are created with malloc and not freed. For example: -(void) memory_leak_bug { struct Person *p = malloc(sizeof(struct Person)); }

Infer: Bug Types

Null dereference

Infer reports null dereference bugs in C, Objective-C and Java. The issue is about a pointer that can be null and it is dereferenced. This leads to a crash in all the above languages.

True Positive: Uninitialized Variable

```
#include<stdio.h>
int main()
{
int  a;
/*Uncomment the below line to fix
the UNINITIALIZED_VALUE issue*/
//a =  __infer_nondet_int();
if (a > 5)
{
a = 1;
}
else
{
a = 2;
}
return a;
}
```

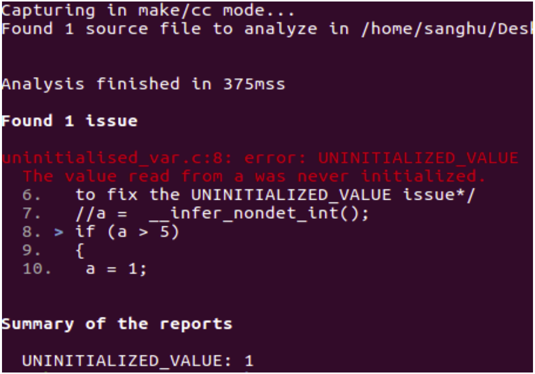
Figure 13: Uninitialized Variable

True Positive: Uninitialized Variable

```
sanghu@paella2018:~/Desktop/Infer/infer-linux64-v0.17.0$
bin/infer run -o uninitialised_var -- clang -c
uninitialised_var.c
```

Figure 14: Uninitialized Variable Command

True Positive: Uninitialized Variable



```
Capturing in make/cc mode...
Found 1 source file to analyze in /home/sanghu/Des

Analysis finished in 375mss

Found 1 issue

uninitialised_var.c:8: error: UNINITIALIZED_VALUE
  The value read from a was never initialized.
  6.    to fix the UNINITIALIZED_VALUE issue*/
  7.    //a =  __infer_nondet_int();
  8. >  if (a > 5)
  9.    {
  10.    a = 1;

Summary of the reports

  UNINITIALIZED_VALUE: 1
```

Figure 15: Uninitialized Variable with issue reported

True Positive: Uninitialized Variable

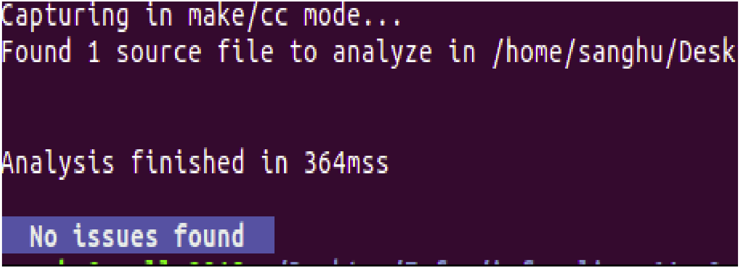


Figure 16: Uninitialized Variable with fixed issue

True Positive: Division by Zero

```
#include<stdio.h>
int main()
{
int  a, b = 0;
a = 8;
if (a > 5)
{

/*Uncomment the below line
to fix the div_by_zero issue
Please add the command option
"--enable-issue-type DIVIDE_BY_ZERO"*/

//b = 1;
a = a / b;
}
else
{
a = 2;
}
return a;
}
```

Figure 17: Division by Zero

True Positive: Division by Zero

```
sanghu@paella2018:~/Desktop/Infer/infer-linux64-v0.17.0$
bin/infer run --enable-issue-type DIVIDE_BY_ZERO
-o div_zero -- clang -c div_zero.c
```

Figure 18: Division by Zero command

True Positive: Division by Zero

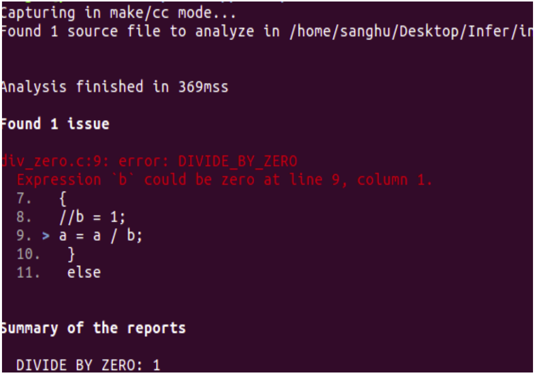


Figure 19: Division by Zero with issue reported

True Positive: Division by Zero

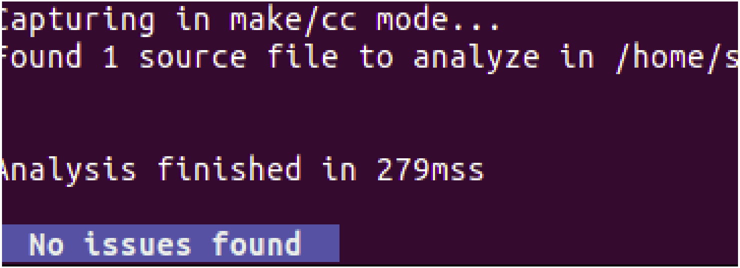


Figure 20: Division by Zero with fixed issue

True Positive: Dead Store

```
#include<stdio.h>
#include<assert.h>
int main()
{
int a,b = 0;
a =  __infer_nondet_int();
if (a > 5)
{
     b = 1;
}
else
{
     b = 2;
}
//Uncomment this line to
//fix the dead_store issue
//return b;
}
```

Figure 21: Dead Store

True Positive: Dead Store

```
sanghu@paella2018:~/Desktop/Infer/infer-linux64-v0.17.0$
bin/infer run -o dead_store -- clang -c dead_store.c
```

Figure 22: Dead Store command

True Positive: Dead Store

```
Capturing in make/cc mode...
Found 1 source file to analyze in /home/sanghu/Desktop/Infer/infer-linux64-v0.17.0/dead_store


Analysis finished in 323mss

Found 2 issues

dead_store.c:9: error: DEAD_STORE
  The value written to &b (type int) is never used.
  7.    if (a > 5)
  8.    {
  9. >      b = 1;
  10.    }
  11.    else

dead_store.c:13: error: DEAD_STORE
  The value written to &b (type int) is never used.
  11.    else
  12.    {
  13. >      b = 2;
  14.    }
  15.    //return b; //Uncomment this line to fix the dead_store issue


Summary of the reports

  DEAD_STORE: 2
```

Figure 23: Dead Store with issue reported

True Positive: Dead Store

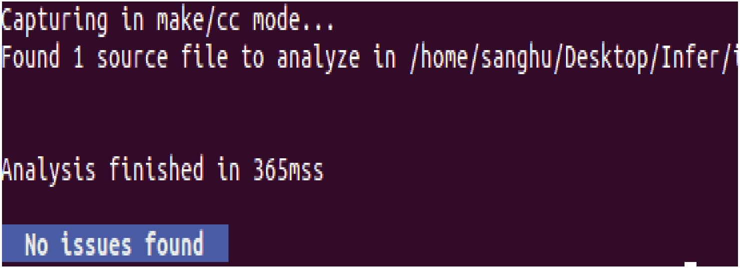


Figure 24: Dead Store with fixed issue

True Positive: Mixtype code

```
#include<stdio.h>
int main()
{
int  a, b = 0;
if (a > 5)
{
a = a / b;
}
else
{
b = 2;
}
return b;
}
```

Figure 25: Mixtype

True Positive: Mixtype code

```
sanghu@paella2018:~/Desktop/Infer/infer-linux64-v0.17.0$
bin/infer run -o mixtype -- clang -c mixtype.c
```

Figure 26: Mixtype command-1

True Positive: Mixtype code

```
Capturing in make/cc mode...
Found 1 source file to analyze in /home/sanghu/Desk

Analysis finished in 404mss

Found 3 issues

mixtype.c:7: error: DEAD_STORE
  The value written to &a (type int) is never used.
  5.     if (a > 5)
  6.     {
  7.  >  a = a / b;
  8.     }
  9.     else

mixtype.c:5: error: UNINITIALIZED_VALUE
  The value read from a was never initialized.
  3.     {
  4.     int  a, b = 0;
  5.  >  if (a > 5)
  6.     {
  7.     a = a / b;

mixtype.c:7: error: UNINITIALIZED_VALUE
  The value read from a was never initialized.
  5.     if (a > 5)
  6.     {
  7.  >  a = a / b;
  8.     }
  9.     else


Summary of the reports

  UNINITIALIZED_VALUE: 2
           DEAD_STORE: 1
```

Figure 27: Mixtype with issue reported

True Positive: Mixtype code

```
sanghu@paella2018:~/Desktop/Infer/infer-linux64-v0.17.0$
bin/infer run --enable-issue-type DIVIDE_BY_ZERO
-disable-issue-type DEAD_STORE -disable-issue-type
 UNINITIALIZED_VALUE -o mixtype -- clang -c mixtype.c
```

Figure 28: Mixtype command-2

True Positive: Mixtype code

```
Capturing in make/cc mode...
Found 1 source file to analyze in /home/sanghu/Deskto


Analysis finished in 413mss

Found 1 issue

mixtype.c:7: error: DIVIDE_BY_ZERO
  Expression `b` could be zero at line 7, column 1.
  5.    if (a > 5)
  6.    {
  7. >  a = a / b;
  8.    }
  9.    else


Summary of the reports

  DIVIDE_BY_ZERO: 1
```

Figure 29: Mixtype with disabled options

Infer: False Positive

```c
/* A sample program for
False Positve case*/
#include <stdio.h>
#include <klee/klee.h>
int main() {
    int x = 0;
    int y = 0;
    int a[1], b[5];
    b[0] = __infer_nondet_int();
    b[1] = __infer_nondet_int();
    b[2] = __infer_nondet_int();
    b[3] = __infer_nondet_int();
    b[4] = __infer_nondet_int();
    int i = 0;
    while(i < 5) {
       if (b[i] > 0)
       x = x + (i+2)*5;
       else x = x + 0;
       if (i == 4 && b[1] > 0)
       x = x + 0;
       else x = x + 30;
       i++;
    }
    int BOUND = 130;
    if (BOUND == x) {y++;}
    a[y] = 5;
    return 0;
}
```

Figure 30: Bound Check via memory error program

Infer: False Positive

```
sanghu@paella2018:~/Desktop/Infer/infer-linux64-v0.17.0$
bin/infer run --bufferoverrun --enable-issue-type
DIVIDE_BY_ZERO --disable-issue-type DEAD_STORE
--disable-issue-type UNINITIALIZED_VALUE -o sample-teaching
 -- clang -c sample-teaching.c
```

Figure 31: Bound Check via memory error command

Infer: False Positive

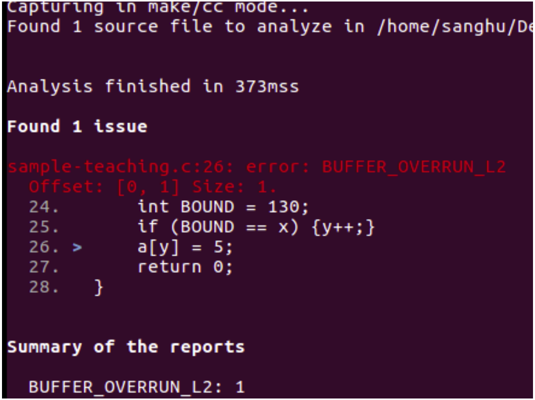


Figure 32: Bound Check via memory error report

Experimental Results

- We have tested ITC-Benchmarks (static analysis benchmarks from Toyota ITC) for C programs
- It can be access here: https://github.com/regehr/itc-benchmarks
- This repository has two categories a) With defects (Injected into programs), and b) Without defects (The same errors have been resolved)
- Defect types related to dynamic memory allocation, error handling, multi-threading

Results

We have tested *defective* benchmarks using Infer Static Analyser to get *POSITIVE*(Bug Found) and *NEGATIVE*(Bug Not-Found) results.

Results

Programs	#TBugs	#POSITIVE	#NEGATIVE
bit_shift.c	17	0	17
buffer_overrun_dynamic.c	32	32	0
buffer_underrun_dynamic.c	40	35	5
cmp_funcadr.c	2	0	2
conflicting_cond.c	10	0	10
data_lost.c	19	0	19
data_overflow.c	25	14	11
data_underflow.c	12	8	4
dead_code.c	11	0	11
dead_lock.c	5	0	5
deletion_of_data_structure_sentinel.c	3	0	3
double_free.c	12	9	3
double_lock.c	5	0	5
double_release.c	5	0	5
endless_loop.c	9	0	9
free_nondynamic_allocated_memory.c	16	16	0
free_null_pointer.c	15	5	10
func_pointer.c	15	0	15
function_return_value_unchecked.c	16	0	16
improper_termination_of_block.c	0	0	0

Table 1: Results-I

Results

Programs	#TBugs	#POSITIVE	#NEGATIVE
insign_code.c	1	0	1
invalid_extern.c	6	0	6
invalid_memory_access.c	17	3	14
littlemem_st.c	11	0	11
livelock.c	0	0	0
lock_never_unlock.c	9	0	9
memory_allocation_failure.c	16	13	3
memory_leak.c	18	11	7
not_return.c	4	0	4
null_pointer.c	17	12	5
overrun_st.c	54	48	6
ow_memcpy.c	2	0	2
pow_related_errors.c	29	0	29
ptr_subtraction.c	2	0	2
race_condition.c	8	0	8
redundant_cond.c	14	0	14
return_local.c	2	0	2
sign_conv.c	19	0	19
sleep_lock.c	3	0	3
st_cross_thread_access.c	7	0	7

Table 2: Results-II

Results

Programs	#TBugs	#POSITIVE	#NEGATIVE
st_overflow.c	7	0	7
st_underrun.c	7	0	7
underrun_st.c	13	7	6
uninit_memory_access.c	15	3	12
uninit_pointer.c	16	4	12
uninit_var.c	15	0	15
unlock_without_lock.c	8	0	8
unused_var.c	7	0	7
wrong_arguments_func_pointer.c	18	2	16
zero_division.c	16	12	4

Table 3: Results-III

Demo on selected programs

- *zero_division.c*
- *bit_shift.c* (Numerical defects)
- *buffer_underrun_dynamic.c* (Dynamic memory defects)
- *func_pointer.c* (Pointer related defects)
- *invalid_memory_access.c* (Resource management defects)
- *memory_leak.c* (Resource management defects)

Thank You!