C/C++ static analysis with LLVM compiler infrastructure

VoIS-Young report

Xiang Chen 2022.03.22

Q1: What is static analysis?

Starts from optimization

```
void twiddle1(long *xp, long *yp) {
    *xp += *yp;
    *xp += *yp;
}
// *xp is scaled by 4

void twiddle2(long *xp, long *yp) {
 *xp += 2* *yp;
}
// *xp is scaled by 3
```

When `xp` and `yp` points to the same address, AKA **pointer aliasing**, this two function act differently.

> "Computer Systems A Programmer's Perspective": Section 5.1 Capabilities and Limitations of Optimizing Compilers P535

From compiler's view

Given the source code of program, compiler does the following things chronically:

- Preprocess: headers and macros, e.g. `clang -E`
- Lexical analysis: get tokens, e.g. `clang -cc1 -dump-tokens`
- Syntax analysis: get AST, e.g. `clang -cc1 -ast-dump -fcolor-diagnostic`
- 🧢 Linter: check code style
- Semantic analysis: type checking on AST
- MiddleEnd: translate to IR, e.g. `clang -emit-llvm -S`
- 🤔 Static analysis: perform analysis on IR
- BackEnd: code generation

> https://cs.nju.edu.cn/tiantan/software-analysis/IR.pdf

From security researcher's view

Given the source code of program, a static analyzer does the following things using a top-down approach:

- Find security issues based on synthetic program properties (variable liveness, feasible path etc)
- Those non-trivial program properties need to be verified by algorithms
- Those algorithms model the program's as mathematic structures (lattice, graph etc)

```c
int main(int argc, char **argv) {
    int x = 0;      // initialization
    if (argc < 0) {
        return 1/x; // not a feasible path!
    }
    int y = 1/x;    // feasible path, divide zero exception!
    x = 1;          // new
    y = 1/x;        // valid division
    return 0;
}
```

> This program is so simple even that it can be analyzed by ChatGPT!, and GPT is capable of finding non-trival bugs.

Finally, the researcher can automate above processes for finding bugs, e.g. taint analysis

Q2: Why focus on C/C++?

C/C++ languege features

- System programming language
  - More than 90% code in Windows and Linux kernel are written in C/C++
  - About 50% code in Google Fuchsia are written in C++
- They don't have secure coding rule at language-level
  - Memory can be manipulated by raw pointer, even smart pointer can be misused (like use-after-move)
  - Implicit cast can cause integer overflow
  - Many secure coding standards has not been widely used
- Complicated source code structure
  - Developers like to define customized memory management function (usually as a wrapper of `malloc`)
  - Context info is limited

Q3: Why use LLVM?

LLVM IR is a great design

- Three forms: memory object / human readable text file (`.ll`) / machine readable bitcode (`.bc`)

> "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation", Chris Lattner and Vikram Adve (CGO '04).

- RISC-style Typed language
- Single Static Assignment (SSA)

```
// source code
int x = 0;
x = x + 1;
// LLVM IR
store i32 0, ptr %2, align 4, !dbg !19
%3 = load i32, ptr %2, align 4, !dbg !20
%4 = add nsw i32 %3, 1, !dbg !21
```

> https://llvm.org/docs/LangRef.html

LLVM has rich libraries

The core of LLVM optimization is LLVM Pass

- Analysis Pass: `-dot-cfg`, `-print-callgraph`, `-domtree`, `-loops`
- Transformation Pass: `-dce`, `-mem2reg`, `-loop-unroll`, `-licm`

> https://www.llvm.org/docs/Passes.html

LLVM also provides many other useful ADT: `StringRef`, `SmallSet`, `BitVector`

> https://www.llvm.org/docs/ProgrammersManual.html

"Let’s feel the power of LLVM!"

Clang Static Analyzer

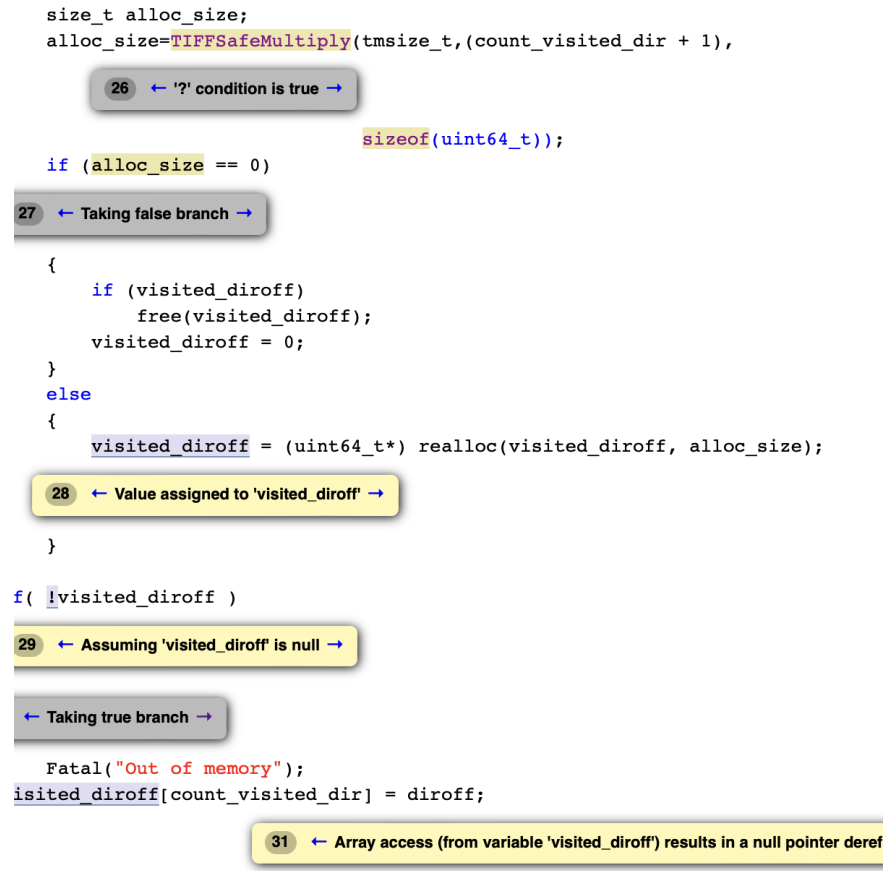
- Command-line usage, for single test

```
$ clang -cc1 -w -fcolor-diagnostics -analyze -analy
/tmp/test.c:6:13: warning: Division by zero [core.D
  int y = 1 / x; // feasible path, divide zero exce
          ~~^~~
1 warning generated.
```

- Scan-build usage, for large project

```
$ scan-build "make -j8"
$ scan-view
```

https://clang-analyzer.llvm.org



Facebook Infer

A tool to detect bugs in Java and C/C++/Objective-C code.

```
$ infer run -- clang /tmp/npd.c
Capturing in make/cc mode...
Found 1 source file to analyze in /infer-out
1/1 [################################################################################] 100% 16.046ms

tmp/npd.c:3: error: Null Dereference
  pointer `p` last assigned on line 2 could be null and is dereferenced at line 3, column 9.
  1. int main() {
  2.     char *p = 0;
  3.     return *p;
              ^
  4. }


Found 1 issue
          Issue Type(ISSUED_TYPE_ID): #
  Null Dereference(NULL_DEREFERENCE): 1
```

https://fbinfer.com

"But there is no silver bullet..."

Q3: Sound or Complete?

Rice Theorem

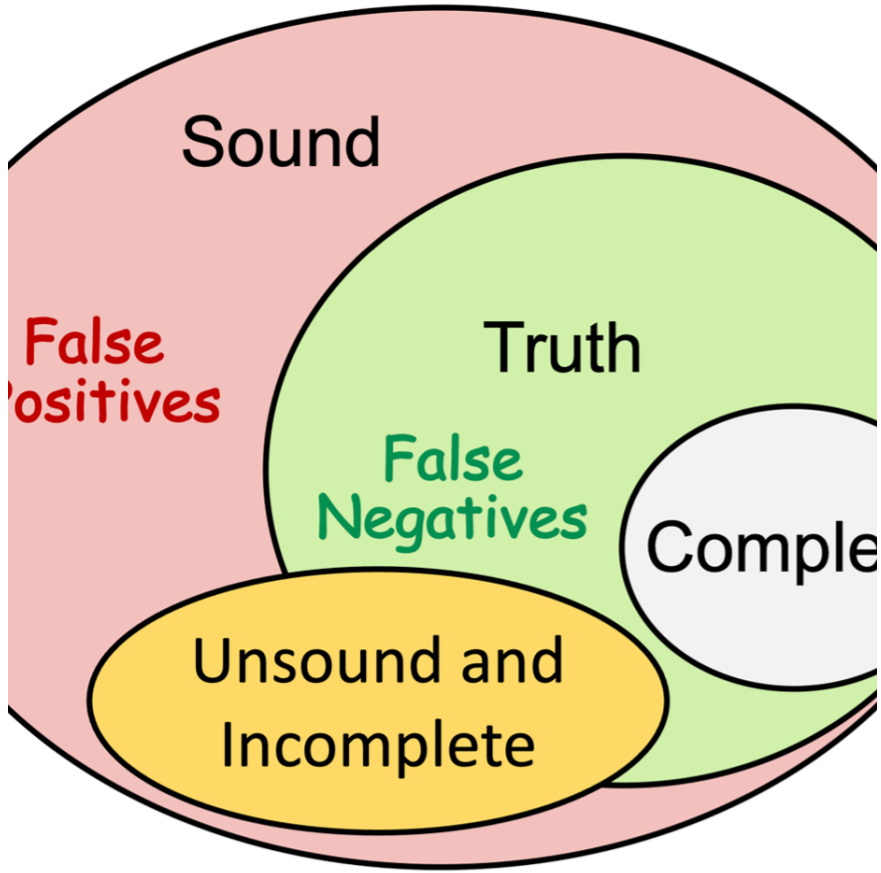
Definition: Any non-trivial property of the behavior of programs in a r.e. language is undecidable.

A property is trivial if either it is not satisfied by any r.e. language, or if it is satisfied by all r.e. languages; otherwise it is non-trivial.

Conclusion: No perfect static analysis, only useful static analysis.

- Compromise soundness (false negatives)
- Compromise completeness (false positives, **preferable to security application**)

> https://cs.nju.edu.cn/tiantan/software-analysis/introduction.pdf

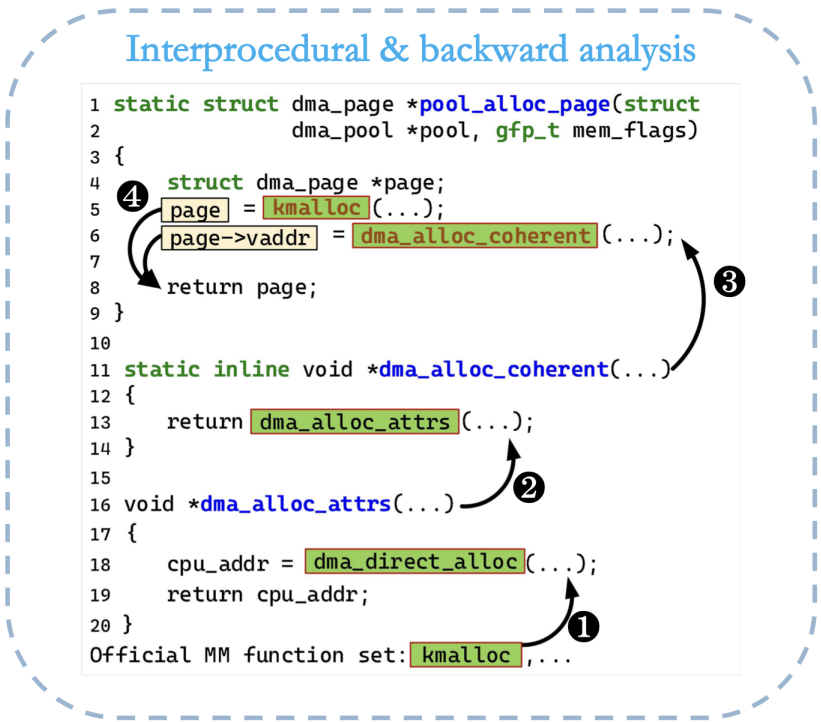


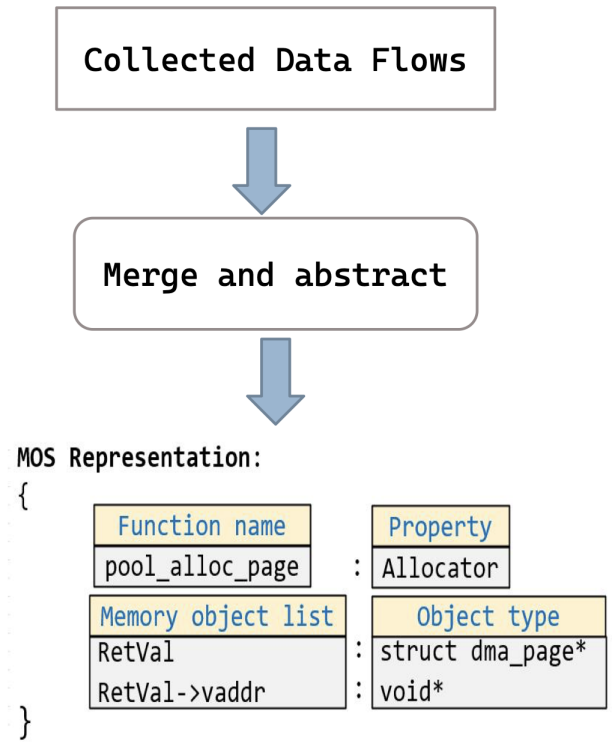
Real World Challenge

- Decrease FN, especially on real-world program
  - Mainstream C/C++ program analysis tools report 47%-80% FN on real-world program
- Decrease FP
  - better solver
  - better modeling for C/C++ semantics

Goshawk

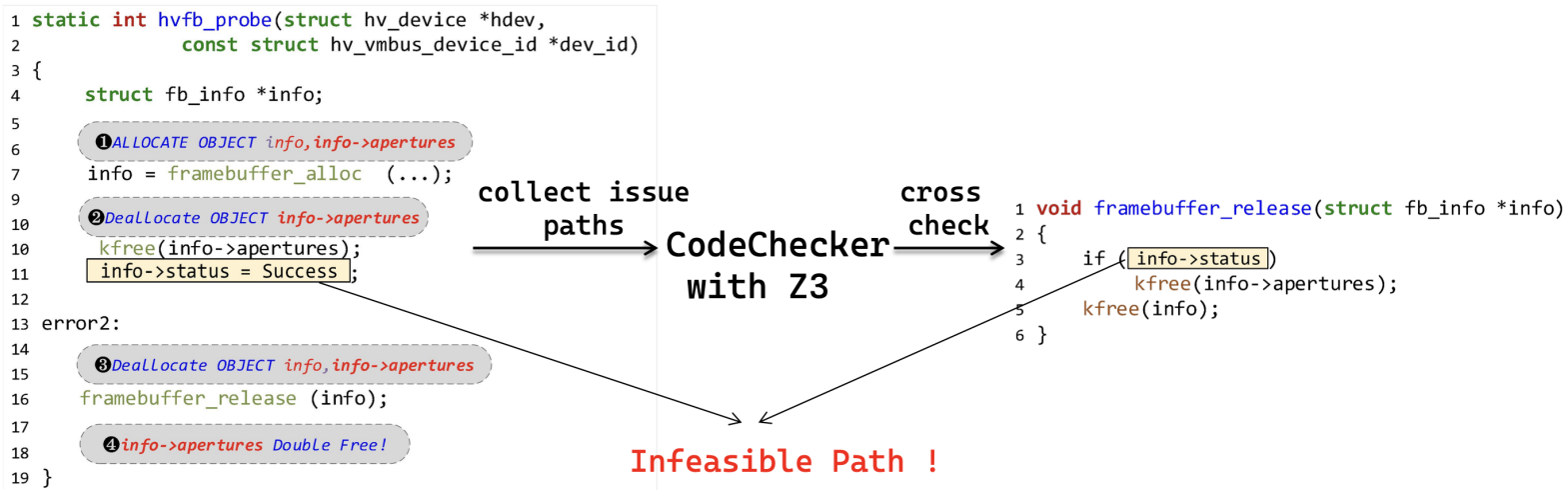
Hunting Memory Corruptions via Structure-Aware and Object-Centric Memory Operation Synopsis





Goshawk

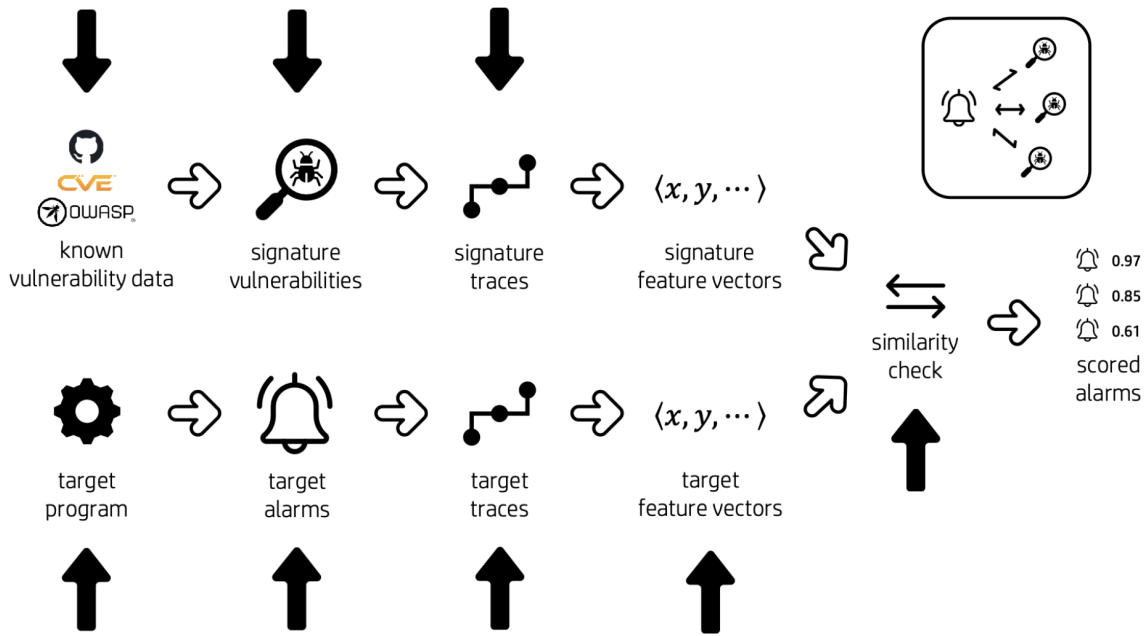
Hunting Memory Corruptions via Structure-Aware and Object-Centric Memory Operation Synopsis



Initial report

TRACER

Signature-based Static Analysis for Detecting Recurring Vulnerabilities



Any question?