

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-diagnostics``
- 🧐 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)

```
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-trivial bugs.

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

Q2: Why focus on C/C++?

C/C++ language 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.DivideZero]
    int y = 1 / x; // feasible path, divide zero except
           ~^~~~
1 warning generated.
```

- Scan-build usage, for large project

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

<https://clang-analyzer.lvm.org>

```
size_t alloc_size;
alloc_size=TIFFSafeMultiply(tmsize_t,(count_visited_dir + 1),
```

26 ← '?' condition is true →

```
sizeof(uint64_t));
```

```
if (alloc_size == 0)
```

27 ← Taking false branch →

```
{
    if (visited_diroff)
        free(visited_diroff);
    visited_diroff = 0;
}
else
{
    visited_diroff = (uint64_t*) realloc(visited_diroff, alloc_size);
```

28 ← Value assigned to 'visited_diroff' →

```
}
```

```
f( !visited_diroff )
```

29 ← Assuming 'visited_diroff' is null →

← Taking true branch →

```
Fatal("Out of memory");
visited_diroff[count_visited_dir] = diroff;
```

31 ← Array access (from variable 'visited_diroff') results in a null pointer deref

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

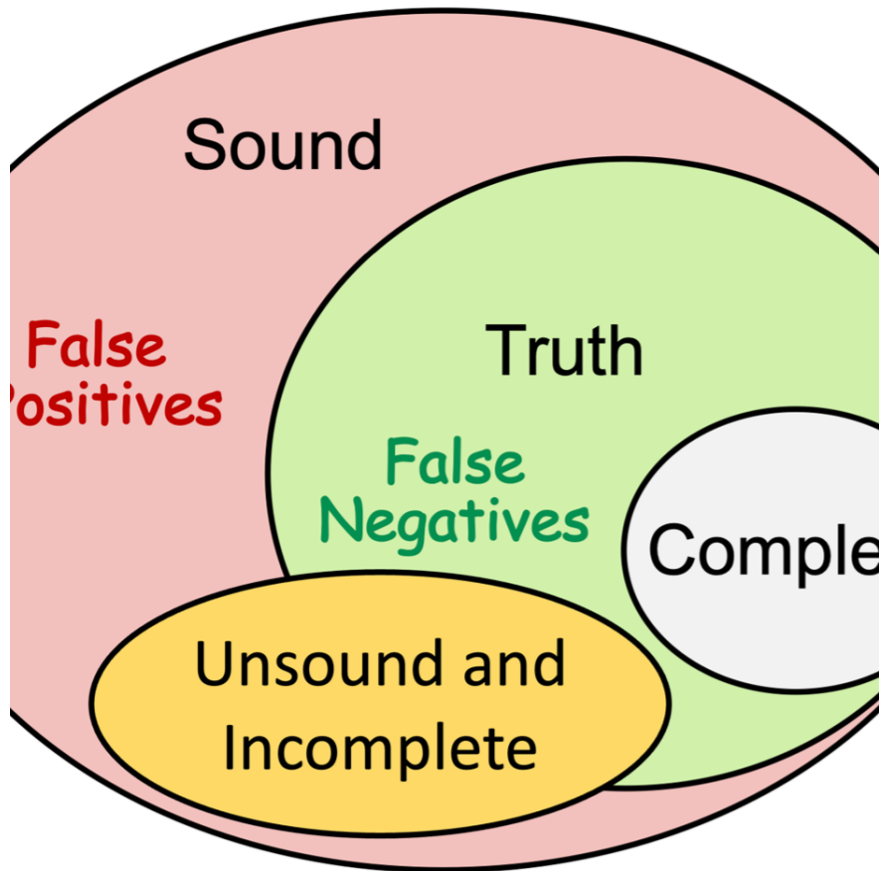
Definition: Any non-trivial property of the behavior of programs in a r.e. language is undecidable.

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

Interprocedural & backward analysis

```
1 static struct dma_page *pool_alloc_page(struct
2     dma_pool *pool, gfp_t mem_flags)
3 {
4     struct dma_page *page;
5     page = kmalloc(...);
6     page->vaddr = dma_alloc_coherent(...);
7     return page;
8 }
9
10
11 static inline void *dma_alloc_coherent(...)
12 {
13     return dma_alloc_attrs(...);
14 }
15
16 void *dma_alloc_attrs(...)
17 {
18     cpu_addr = dma_direct_alloc(...);
19     return cpu_addr;
20 }
21
22 Official MM function set: kmalloc, ...
```

Diagram illustrating interprocedural and backward analysis. The code snippet shows the flow of memory allocation and return values. Arrows indicate the flow of data and control flow between functions:

- ①: Call to `kmalloc` from `dma_alloc_attrs`.
- ②: Call to `dma_alloc_attrs` from `dma_alloc_coherent`.
- ③: Call to `dma_alloc_coherent` from `pool_alloc_page`.
- ④: Call to `pool_alloc_page` from the caller.

Collected Data Flows



Merge and abstract



MOS Representation:

{	
Function name	Property
pool_alloc_page	: Allocator
Memory object list	Object type
RetVal	: struct dma_page*
RetVal->vaddr	: void*
}	

Goshawk

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

```
1 static int hvfb_probe(struct hv_device *hdev,  
2                       const struct hv_vmbus_device_id *dev_id)  
3 {  
4     struct fb_info *info;  
5  
6     ❶ ALLOCATE OBJECT info, info->apertures  
7     info = framebuffer_alloc (...);  
8  
9     ❷ Deallocate OBJECT info->apertures  
10    kfree(info->apertures);  
11    info->status = Success;  
12  
13 error2:  
14  
15    ❸ Deallocate OBJECT info, info->apertures  
16    framebuffer_release (info);  
17  
18    ❹ info->apertures Double Free!  
19 }
```

collect issue
paths

CodeChecker
with Z3

cross
check

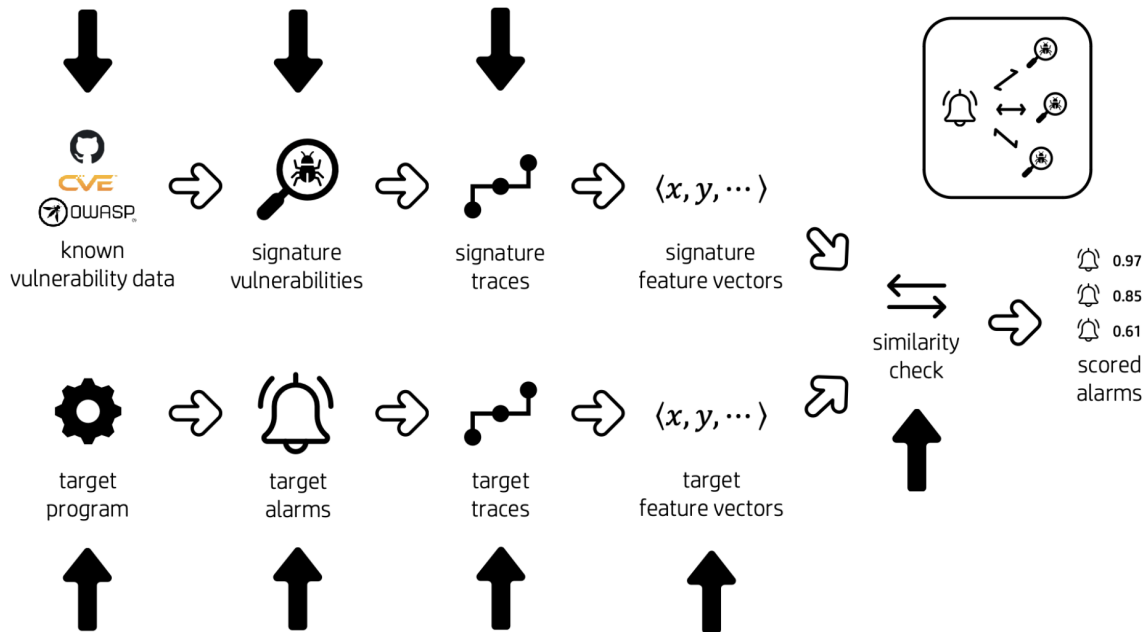
```
1 void framebuffer_release(struct fb_info *info)  
2 {  
3     if (info->status)  
4         kfree(info->apertures);  
5     kfree(info);  
6 }
```

Infeasible Path !

Initial report

TRACER

Signature-based Static Analysis for Detecting Recurring Vulnerabilities



Any question?