Symbiotic with CPAchecker

Marek Chalupa, Masaryk University Brno
October 2, 2019

4th International Workshop on CPAchecker
```c
int main(void)
{
    int x = 10;
    for (int i = 0; i < 1000; ++i) {
        for (int j = 0; j < 1000; ++j) {
            for (int k = 0; k < 1000; ++k) {

                /* some code that does not touch x */
            }
        }
    }
    if (x > 0)
        __VERIFIER_error();
}
```
int main(void)
{
    int x = 10;

    if (x > 0)
        __VERIFIER_error();
}

int main(void)
{
  int x = 10;
  for (int i = 0; i < 1000; ++i) {
    for (int j = 0; j < 1000; ++j) {
      for (int k = 0; k < 1000; ++k) {
/* some code that does not touch x */
      }
    }
  }
  if (x > 0)
    __VERIFIER_error();
}

__VERIFIER_error();
• Symbiotic
  ● What is Symbiotic?
  ● How it works?

• CPAchecker in Symbiotic
  ● How is it integrated?
  ● Some results.
Symbiotic
What is Symbiotic?

- Framework for generating code fitted to verification.
What is Symbiotic?

- Framework for generating code fitted to verification.
- It employs:
  - code instrumentation (combined with static analyses),
  - program slicing,
  - (compiler) optimizations.
- It is highly modular.
- Internally works with LLVM.
- Integrates several verification tools that can be seamlessly run on the generated code.
What is Symbiotic?

- Framework for generating code fitted to verification.
- It employs:
  - code instrumentation (combined with static analyses),
  - program slicing,
  - (compiler) optimizations.
- It is highly modular.
- Internally works with LLVM.
What is Symbiotic?

- Framework for generating code fitted to verification.
- It employs:
  - code instrumentation (combined with static analyses),
  - program slicing,
  - (compiler) optimizations.
- It is highly modular.
- Internally works with LLVM.
- Integrates several verification tools that can be seamlessly run on the generated code.
Symbiotic – schema

Compilation

CLANG

Instrumentation

Optimizations

Slicing

Optimizations

Verification

KLEE, CPAchecker, ...

Symbiotic-cc

Symbiotic-verify
Instrumentation inserts auxiliary code to the analyzed program.

- Symbiotic has a configurable instrumentation module
- Configuration in JSON
Instrumentation inserts auxiliary code to the analyzed program.

- Symbiotic has a configurable instrumentation module
- Configuration in JSON
- It can use results of static analyses to prevent redundant code injection
- Can run in several stages, passing information from former to later stages
Instrumentation

Instrumentation inserts auxiliary code to the analyzed program.

- Symbiotic has a configurable instrumentation module
- Configuration in JSON
- It can use results of static analyses to prevent redundant code injection
- Can run in several stages, passing information from former to later stages
- Restriction: can insert only calls to functions at this moment
1. \%p = alloca i32*
   call check_pointer(\%p, 8)
2. store null to \%p
3. \%addr = call malloc(20)
   call check_pointer(\%p, 8)
4. store \%addr to \%p
   call check_free(\%addr)
5. call free(\%addr);
   call check_pointer(\%p, 8)
6. \%tmp = load \%p
   call check_pointer(\%tmp, 4)
7. store i32 1 to \%tmp
Instrumentation – example

1. %p = alloca i32*

2. store null to %p

3. %addr = call malloc(20)

4. store %addr to %p

5. call free(%addr);

6. %tmp = load %p
   call check_pointer(%tmp, 4)

7. store i32 1 to %tmp
Program slicing removes instructions of a program that are irrelevant to a specified "behavior" of the program.

- The behavior is specified by slicing criterion \(< V, l >\)
  - \(V\) is a set of variables
  - \(l\) is a program location
  - meaning: preserve the value of variables in \(V\) at location \(l\) (and the reachability of \(l\)) during any execution of the program
- Slicing criteria are (in our settings) error locations
Program slicing – how it works?

- Compute dependencies between instructions.
- We say that instruction A depends on instruction B if:
  - instruction A uses values generated by instruction B, or
  - instruction A is not executed if we go some other way at (branching) B.
- Slicing: keep only the instructions on which the error (transitively) depends.
Program slicing – how it works?

- Compute dependencies between instructions.
- We say that instruction A depends on instruction B if:
  - instruction A uses values generated by instruction B, or
  - instruction A is not executed if we go some other way at (branching) B.
- Slicing: keep only the instructions on which the error (transitively) depends.
Program slicing – how it works?

- Compute dependencies between instructions.
  - We say that instruction A depends on instruction B if:
    - instruction A uses values generated by instruction B, or
    - instruction A is not executed if we go some other way at (branching) B.
  - Slicing: keep only the instructions on which the error (transitively) depends.
int zeroing(char *buf, size_t size)
{
    int n = input();
    for (int i = 0; i < n; ++i) {
        assert(i < size && "Out of bounds");
        buf[i] = 0;
    }
}

return 0;
}
int zeroing(char *buf, size_t size)
{
    int n = input();
    for (int i = 0; i < n; ++i) {
        assert(i < size && "Out of bounds");
        buf[i] = 0;
    }
}

return 0;
Once the code is generated, we can

- Do nothing... (output the generated LLVM)
- Generate C from it and output it (llvm2c tool)
- Pass it to a verification engine (as LLVM or C, according to the verifier)
Once the code is generated, we can

- Do nothing... (output the generated LLVM)
- Generate C from it and output it (llvm2c tool)
- Pass it to a verification engine (as LLVM or C, according to the verifier)

So Symbiotic can be viewed as a

- C to LLVM compiler,
- C to C transformer,
- verification tool for C language
Verification engines

- Verification tools are integrated into Symbiotic by extending benchexec tool-info modules.
- The extension adds methods for:
  - specifying the required LLVM version
  - (optional) setting the environment
  - (optional) hooks that run before or after compilation/instrumentation/slicing/verification

So far we have integrated KLEE, CPAchecker, DIVINE, SMACK, and SeaHorn. Experimental support for CBMC, UltimateAutomizer, and IKOS.
Verification engines

- Verification tools are integrated into Symbiotic by extending benchexec tool-info modules
- The extension adds methods for:
  - specifying the required LLVM version
  - (optional) setting the environment
  - (optional) hooks that run before or after compilation/instrumentation/slicing/verification
- So far we have integrated KLEE, CPAchecker, DIVINE, SMACK, and SeaHorn
  - experimental support for CBMC, UltimateAutomizer, and IKOS
Symbiotic – Limits.

- No C++ (exceptions).
- Symbiotic still does not scale to large programs.
  - The current bottle-neck is data-dependence analysis in slicer.
Symbiotic with CPAchecker
CPAchecker integration into Symbiotic

- CPAchecker has LLVM backend
  - Parses LLVM and creates a CFA over C language
  - Missing a support for some floats-related constructs
  - Missing a support for some global initializers
• CPAchecker has LLVM backend
  • Parses LLVM and creates a CFA over C language
  • Missing a support for some floats-related constructs
  • Missing a support for some global initializers
• We can use also the C backend directly
  • Symbiotic can use llvm2c to generate C from the LLVM
  • The default option (as of a few weeks ago :)
CPAchecker integration into Symbiotic

- CPAchecker has LLVM backend
  - Parses LLVM and creates a CFA over C language
  - Missing a support for some floats-related constructs
  - Missing a support for some global initializers
- We can use also the C backend directly
  - Symbiotic can use llvm2c to generate C from the LLVM
  - The default option (as of a few weeks ago :)
- We use the SV-COMP'19 configuration (-svcomp19) by default
CPAchecker in Symbiotic

- Symbiotic + llvm2c + CPAchecker (with the C backend) now works better than Symbiotic + CPAchecker (LLVM backend)
- However, "pure" CPAchecker still works better than Symbiotic + CPAchecker
Experiments on ReachSafety category
Experiments with LLVM backend

![Graph showing CPU time vs n-th fastest benchmark for CPAchecker and KLEE with and without slicing.]
Symbiotic is a framework that generates optimized (LLVM or C) code for verification.

- It is highly modular.
- It integrates several verification tools, including CPAchecker.

https://github.com/staticafi/symbiotic
Summary

- Symbiotic is a framework that generates optimized (LLVM or C) code for verification
- It is highly modular
- It integrates several verification tools, including CPAchecker

https://github.com/staticafi/symbiotic

Thank you!