Oracle-Guided Component-Based Program Synthesis

Susmit Jha  Sumit Gulwani  Sanjit Seshia  Ashish Tiwari

UC Berkeley  MSR  UC Berkeley  SRI
From Verification to Synthesis

Synthesizing systems that are correct-by-construction has always been a holy grail for computer science.
From Verification to Synthesis

*Synthesizing systems that are correct-by-construction has always been a holy grail for computer science.*

E. M. Clarke and E. A. Emerson, 1981:

“We propose a method of constructing concurrent programs in which the *synchronization skeleton of the program is automatically synthesized* from a high-level (branching time) Temporal Logic specification.”

1st sentence of their original model checking paper
Turing Award 2007

**Verification-guided Synthesis Approach:** Sketch (ASPLOS06), VS3(POPL10)
From Verification to Synthesis

- Current methods:
  Verification scalable $\Rightarrow$ Synthesis feasible

- Verification relies on usable complete formal specification.

- What if such a specification is not available?
An Example Application

Deobfuscating Malware
Deobfuscation

int obfuscated (int y) {
    a=1; b=0; z=1; c=0;
    while(1) {
        if (a == 0) {
            if (b == 0) { y=z+y; a =~a; b=~b; c=~c; if (~c) break; }
            else {
                z=z+y; a=~a; b=~b; c=~c; if (~c) break; }
        } else if (b == 0) { z=y << 2; a=~a; }
        else { z=y << 3; a=~a; b=~b; }
    }
}

CONFICKER WORM
discovered in early November 2008, infected ~15 million computers by January, 2009
Deobfuscation

int obfuscated (int y) {
    a=1; b=0; z=1; c=0;
    while(1) {
        if (a == 0) {
            if (b == 0) { y=z+y; a =~a; b=~b; c=~c; if (~c) break; }
            else {
                z=z+y; a=~a; b=~b; c=~c; if (~c) break; }
        }
        else if(b == 0) {z=y << 2; a=~a;}
        else { z=y << 3; a=~a; b=~b; }
    }
}

CONFICKER WORM

discovered in early November 2008, infected ~15 million computers by January, 2009

int multiply45 (int y) {
    z = y << 2;
    y = z + y;
    z = y << 3;
    y = z + y;
    return y; }

Only Specification: Obfuscated Code, created by attacker
Question

Can we automate synthesis in the absence of usable formal specifications?
Main Ideas

Can we automate synthesis in the absence of usable formal specifications? **YES**

- **Program** = Composition of component functions
- **Specification** = Input/output oracle
- **Program Synthesis** = Learning from examples
Contributions

Can we automate synthesis in the absence of usable formal specifications?  YES

- Program = Composition of component functions
  - Efficient encoding of compositions into SAT (SMT)

- Specification = Input/output oracle
  - No need for complete functional specification

- Program Synthesis = Learning from examples
  - Distinguishing inputs for efficient learning
Outline

- Problem Definition
- Approach
- Results
- Related Work
- Contributions
- Conclusion
Class of programs

- Programs implementing functions: \( \mathbf{I} \rightarrow \mathbf{O} \)

\[
P(\mathbf{I}): \\
O_1 = f_{\pi_1}(V_{\pi_1}) \\
O_2 = f_{\pi_2}(V_{\pi_2}) \\
\ldots \\
O_n = f_{\pi_n}(V_{\pi_n})
\]

Where,

- \( \pi_1, \pi_2, \ldots, \pi_n \) is permutation of \( \{1, 2, \ldots, n\} \)
- \( f_{\pi_1}, f_{\pi_2}, \ldots, f_{\pi_n} \) are functions from a given component library

Functions could be if-then-else definitions and hence, the above represents any loop-free code.
Deobfuscation

**multiply45**: int \( \rightarrow \) int

```
int multiply45 (int y) {
  o1 = f1 (y,2);
  o2 = f2 (o1,y);
  o3 = f1 (o3,3);
  o4 = f2 (o3,o2);
  return o4;
}
```

Component Library

- \( f_1(x,y) : x \ll y \)
- \( f_2(x,y) : x + y \)

```
int multiply45 (int y) {
  z = y \ll 2;
  y = z + y;
  z = y \ll 3;
  y = z + y;
  return y;
}
```
Another Application: Bit-manipulation Programs

- **Loop-free** code to manipulate bit-vectors
  - bitwise operators such as ||, &&, +, -

- **Challenge?**
  - *Short* but very *unintuitive*
  - Often used in *performance critical* code such as network routers, embedded systems
    - Avoid branching for better cache performance
    - Minimize instruction length
TOY EXAMPLE: Turn off rightmost contiguous 1 bits
10110 → 10000, 11010 → 11000

TurnoffRmOnes (x) {
    i = length(x) - 1;
    while( x[i] == 0 ){
        i--; if (i < 0) return x;
    }
    x[i] = 0; i--;
    while( x[i] == 1 ){
        x[i] = 0; i--; if (i < 0)
        return x;
    }
    return x;
}

Naïve loopy implementation

Using bit-vector operators
Components

\[
\text{TurnoffRmOnes} (x) \{ \\
\quad r_1 = x - 1; \\
\quad r_2 = x || r_1; \\
\quad r_3 = r_2 + 1; \\
\quad r_4 = r_3 && x \\
\quad \text{return } r_4; \\
\}\]

Sufficient components for correct program

Extra Components
TurnoffRmOnes (x) {
    r_1 = x - 1;
    r_2 = x || r_1;
    r_3 = r_2 + 1;
    r_4 = r_3 && x
    return r_4;
}
SomethingElse \((x)\) {
    \(r_1 = x - 1;\)
    \(r_5 = !x\)
    \(r_2 = r_5 \text{ || } r_1;\)
    \(r_4 = r_2 \&\& r_5;\)
    return \(r_4;\)
}
Composition

Some composition topology do not represent a valid program

Wrong (x) {
  r_1 = x - 1;
  r_2 = x || r_3;
  r_3 = r_2 + 1;
  r_4 = r_3 && x
  return r_4;
}

UNDEFINED VAR ERROR!
Synthesis Using SMT

• The space of all possible programs obtained by well-formed composition of components expressed as satisfiability formula

\[ \text{Behave}(I, O, L) \]

- \( I \) are inputs to program,
- \( O \) is output of program,
- \( L \) are composition variables

• Satisfiability Modulo Theory
  - A formula in first-order logic
  - some function and predicate symbols have additional interpretations
  - Beaver (UC Berkeley), CVC (NYU), Yices (SRI), Z3 (MSR)
Two key subroutines

- Discovering a program $P$ consistent with some set of input/output examples $E$.

- Discovering a semantically different program $P'$ consistent with set $E$ and an input $I'$ on which $P$ and $P'$ differ.
Synthesis Using SMT

For given set of input, output examples $E$, a consistent program is obtained by satisfiability solving:

$$\exists \ L \ . \ \bigwedge_k \text{Behave}(i_k, o_k, L) \text{ for all } (i_k, o_k) \in E$$

Examples $E$

Solve for $L$

Composition $L$ that works for all examples $E$
Distinguishing Input

Examples E

Composition $L'$ that works for all examples $E$ and differs from $L$ on at least one input $I'$

Solve for $I', L'$

$I_1$, $O_1$  $I_2$, $O_2$  $I_n$, $O_n$
Approach

Space of all possible programs
Each dot represents semantically unique program
Example I/O set $E := \{(i_1, o_1)\}$

Space of all possible programs
Example I/O set $E := \{(i_1,o_1)\}$

Space of all possible programs
Approach

Example I/O set $E := \{(i_1, o_1)\}$

Space of all possible programs
Approach

Example I/O set $E := \{(i_1, o_1)\}$

Space of all possible programs
Example I/O set $E := E \cup \{(i_2, o_2)\}$

Space of all possible programs
Example I/O set $E := E \cup \{(i_j,o_j)\}$

Space of all possible programs
Approach

Example I/O set $E := E \cup \{(i_k, o_k)\}$

Space of all possible programs
Approach

Example I/O set \( E := E \cup \{(i_n, o_n)\} \)

Space of all possible programs
Correctness

Library of components is sufficient? [YES/NO]

I/O pairs show infeasibility? [YES/NO]

Correct design

Infeasibility reported

Incorrect design
Result Highlights

- Applied the technique for deobfuscating pieces of code from
  - Conficker worm
  - MyDoom and
  - survey paper on obfuscations by Collberg et al*

- Synthesized over 35 bit-manipulation programs from Hackers’s delight – Bible of bit-manipulation.

- Program length: 3-15

- Number of input/output examples: 2 to 13.

- Total runtime: < 1 second to 5 minutes.

# Performance

| Program  | Program length # of components | Random Input Runtime (seconds) | SILVER Runtime (seconds) | SILVER Iterations (|E|) |
|----------|---------------------------------|-------------------------------|--------------------------|----------------------|
| P18 (bv) | 4                               | 228.8                         | 25.5                     | 6                    |
| P19 (bv) | 4                               | 163.8                         | 65.5                     | 7                    |
| P20 (bv) | 6                               | 214.1                         | 63.2                     | 8                    |
| P21 (bv) | 7                               | 1074.0                        | 272.3                    | 8                    |
| P22 (bv) | 12                              | timeout                       | 185.6                    | 9                    |
| P23 (deobf) | 3                            | 24.3                          | 1.4                      | 3                    |
| P24 (deobf) | 4                            | 12.0                          | 5.3                      | 2                    |

*Scalability depends on # of component functions and their complexity*
Related Work

  - Full functional specification required
  - Program synthesized from proof
  - Based on deductive reasoning: hard to automate.

- **Superoptimization AHA (Massalin, ASPLOS 87)**
  - Given straight-line sequence of instructions.
  - Generate smaller and equivalent sequence.
  - Enumerate sequence of instructions of increasing length.
  - Use random inputs to prune out wrong programs (unsound).

- **Denali (Combines the above two approaches)**

- **SKETCH (Solar-Lezama et al, ASPLOS 06)**
  - Full functional specification
  - A template/scaffold for target implementation
  - Verification-guided combinatorial search over space of template completions.
Contributions

- An interactive program synthesis approach without in-the-loop verification

- New program learning technique based on the idea of using distinguishing inputs.

- Efficient encoding of program synthesis as satisfiability constraints.

- Applied to deobfuscation of snippets of malware code and synthesis of bit-manipulation programs.
Conclusion

- **SILVER** (*Synthesis Using Integrated Learning and Verification*) can be used for automated program synthesis.
  - discovering short but unintuitive programs
  - program understanding by discovering simpler equivalent program.
Thanks!

- **SILVER** (*Synthesis Using Integrated Learning and Verification*) can be used for automated program synthesis.

- It can aid developers and designers by discovering short but unintuitive programs.

- It can be used for program understanding by discovering simpler equivalent program.
Thanks

Questions ?