# Concolic Execution (Lecture 5 part 2)

Prof. Mathy Vanhoef
DistriNet – KU Leuven – Belgium

# Concolic (concrete + symbolic) execution

Execute the program normally but still gather path constraints

- That is, do **concrete and symbolic execution in parallel**
- Explore one path at a time, from beginning to end
- The concrete input "defines" which path is taken

After an execution, negate a branch decision, and re-execute with new input that triggers the other branch decision

- This new concrete input will follow a different path
- Also called **dynamic symbolic execution**

# Dynamic Symbolic Execution (DSE)

```
int double(int n) {
  return 2 * n;
}

void f(int x, int y){
  int z = double(y);
  if (z == x) {
    if (x > y + 10) {
      assert(0);
    }
  }
}
```

| Concrete execution | Symbolic execution | |
|---|---|---|
| Concrete state | Symbolic state | Path constraint |
| x = 22 | x = $\alpha_1$ | |
| y = 7 | y = $\alpha_2$ | |

# Dynamic Symbolic Execution (DSE)

```
int double(int n) {
  return 2 * n;
}

void f(int x, int y){
  int z = double(y);
  if (z == x) {
    if (x > y + 10) {
      assert(0);
    }
  }
}
```

| Concrete execution | Symbolic execution | |
|---|---|---|
| Concrete state | Symbolic state | Path constraint |
| x = 22 | x = $\alpha_1$ | |
| y = 7 | y = $\alpha_2$ | |
| z = 14 | z = $2 * \alpha_2$ | |

# Dynamic Symbolic Execution (DSE)

```
int double(int n) {
  return 2 * n;
}

void f(int x, int y){
  int z = double(y);
  if (z == x) {
    if (x > y + 10) {
      assert(0);
    }
  }
}
```

| Concrete execution | Symbolic execution | |
|---|---|---|
| Concrete state | Symbolic state | Path constraint |
| x = 22 | x = $\alpha_1$ | $2 * \alpha_2 \neq \alpha_1$ |
| y = 7 | y = $\alpha_2$ | |
| z = 14 | z = $2 * \alpha_2$ | |

- Take the path constraint and negate a branch decision: $2 * \alpha_2 = \alpha_1$
- Solution: $\alpha_1 = 2,\ \alpha_2 = 1$

# Dynamic Symbolic Execution (DSE)

```
int double(int n) {
  return 2 * n;
}

void f(int x, int y){
  int z = double(y);
  if (z == x) {
    if (x > y + 10) {
      assert(0);
    }
  }
}
```

| Concrete execution | Symbolic execution | |
|---|---|---|
| Concrete state | Symbolic state | Path constraint |
| x = 2 | x = $\alpha_1$ | |
| y = 1 | y = $\alpha_2$ | |
| | | |

# Dynamic Symbolic Execution (DSE)

```
int double(int n) {
  return 2 * n;
}

void f(int x, int y){
  int z = double(y);
  if (z == x) {
    if (x > y + 10) {
      assert(0);
    }
  }
}
```

| Concrete execution | Symbolic execution | |
|---|---|---|
| Concrete state | Symbolic state | Path constraint |
| x = 2 | x = $\alpha_1$ | |
| y = 1 | y = $\alpha_2$ | |
| z = 2 | z = $2 * \alpha_2$ | |

# Dynamic Symbolic Execution (DSE)

```
int double(int n) {
  return 2 * n;
}

void f(int x, int y){
  int z = double(y);
  if (z == x) {
    if (x > y + 10) {
      assert(0);
    }
  }
}
```

| Concrete execution | Symbolic execution | |
|---|---|---|
| Concrete state | Symbolic state | Path constraint |
| x = 2 | x = $\alpha_1$ | $2 * \alpha_2 = \alpha_1$ |
| y = 1 | y = $\alpha_2$ | |
| z = 2 | z = $2 * \alpha_2$ | |

# Dynamic Symbolic Execution (DSE)

```
int double(int n) {
  return 2 * n;
}

void f(int x, int y){
  int z = double(y);
  if (z == x) {
    if (x > y + 10) {
      assert(0);
    }
  }
}
```

| Concrete execution | Symbolic execution | |
|---|---|---|
| Concrete state | Symbolic state | Path constraint |
| x = 2 | x = $\alpha_1$ | $2 * \alpha_2 = \alpha_1$ |
| y = 1 | y = $\alpha_2$ | $\alpha_1 \leq \alpha_2 + 10$ |
| z = 2 | z = $2 * \alpha_2$ | |

- Take the path constraint & negate a branch decision: $2 * \alpha_2 = \alpha_1 \land \alpha_1 > \alpha_2 + 10$
- Solution: $\alpha_1 = 30,\ \alpha_2 = 15$

# Dynamic Symbolic Execution (DSE)

```
int double(int n) {
  return 2 * n;
}

void f(int x, int y){
  int z = double(y);
  if (z == x) {
    if (x > y + 10) {
      assert(0);
    }
  }
}
```

| Concrete execution | Symbolic execution | |
|---|---|---|
| Concrete state | Symbolic state | Path constraint |
| x = 30 | x = $\alpha_1$ | |
| y = 15 | y = $\alpha_2$ | |
| | | |

# Dynamic Symbolic Execution (DSE)

```
int double(int n) {
  return 2 * n;
}

void f(int x, int y){
  int z = double(y);
  if (z == x) {
    if (x > y + 10) {
      assert(0);
    }
  }
}
```

| Concrete execution | Symbolic execution | |
|---|---|---|
| Concrete state | Symbolic state | Path constraint |
| x = 30 | x = $\alpha_1$ | |
| y = 15 | y = $\alpha_2$ | |
| z = 30 | x = $2 * \alpha_2$ | |

# Dynamic Symbolic Execution (DSE)

```
int double(int n) {
  return 2 * n;
}

void f(int x, int y){
  int z = double(y);
  if (z == x) {
    if (x > y + 10) {
      assert(0);
    }
  }
}
```

| Concrete execution | Symbolic execution | |
|---|---|---|
| Concrete state | Symbolic state | Path constraint |
| x = 30 | x = $\alpha_1$ | $2 * \alpha_2 = \alpha_1$ |
| y = 15 | y = $\alpha_2$ | |
| z = 30 | x = $2 * \alpha_2$ | |

# Dynamic Symbolic Execution (DSE)

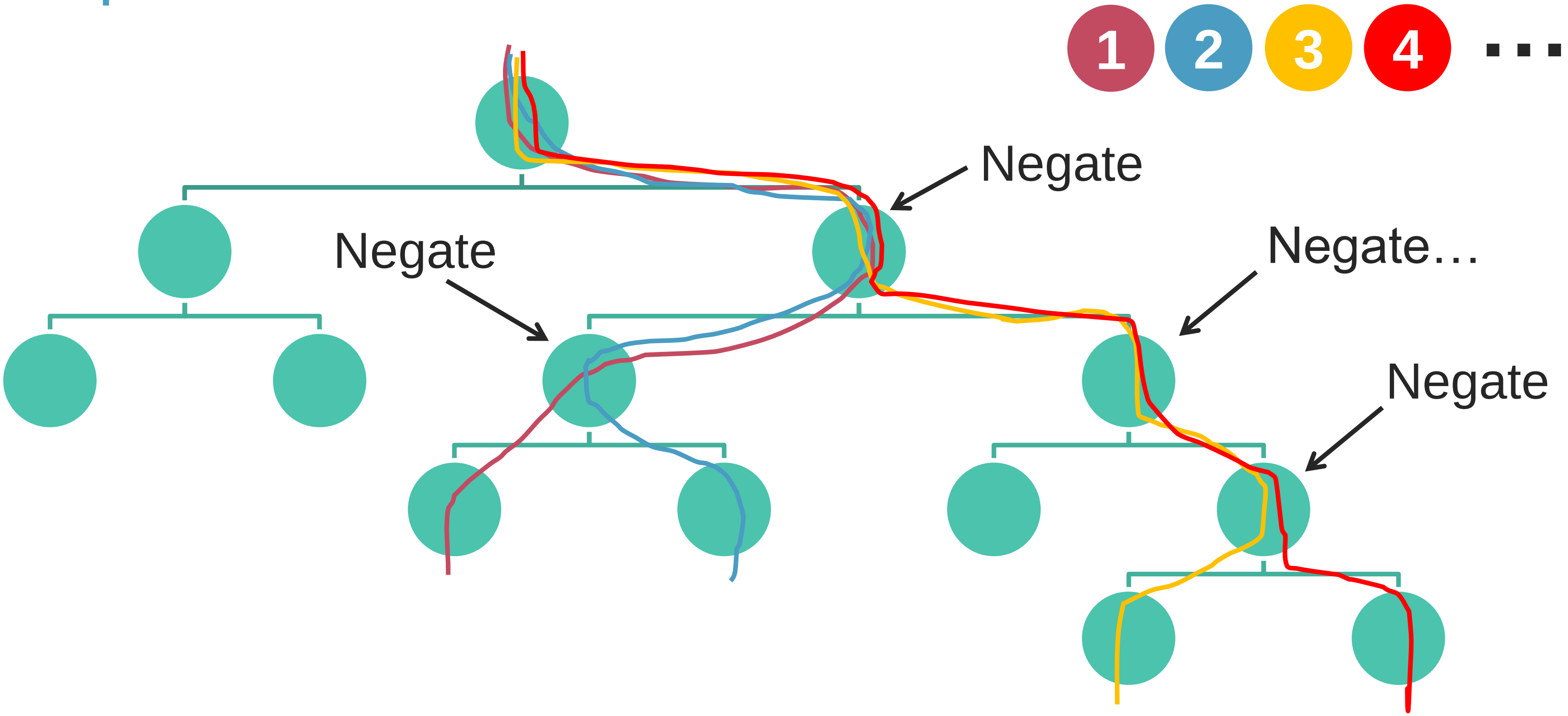
```
int double(int n) {
  return 2 * n;
}

void f(int x, int y){
  int z = double(y);
  if (z == x) {
    if (x > y + 10) {
      assert(0);
    }
  }
}
```

| Concrete execution | Symbolic execution | |
|---|---|---|
| Concrete state | Symbolic state | Path constraint |
| x = 30 | x = $\alpha_1$ | $2 * \alpha_2 = \alpha_1$ |
| y = 15 | y = $\alpha_2$ | $\alpha_1 > \alpha_2 + 10$ |
| z = 30 | x = $2 * \alpha_2$ | |

→ Program crashes, vulnerability has been detected!

# Exploration of the execution tree



# High-level DSE Algorithm

Repeat until all paths are covered:

1. Run program with concrete input $\mathrm{i}$ & collect path constraints $C$
2. Negate any branch condition in the path constraint to take another branch $b'$ → constraints $C'$
3. Call SMT solver to find solution for $C'$ : new concrete input $\mathrm{i}'$
4. Execute program with new input $\mathrm{i}'$ to take branch $b'$
5. Check that $b'$ is indeed taken (i.e., detect non-determinism)

# Advantage and disadvantages of DSE

When the SMT solver can’t handle the constraints (they are too complex) we can easily fall back to concrete values

- Can also use to handle operations not supported by the solver (e.g., floating point operations)
- And can concretize when calling native/system/OS functions

Downside of concretization: analysis is no longer complete

- That is, not all possible paths might get explored

# Dynamic symbolic execution engines

SAGE (symbolic execution for x86)

- Internal Microsoft tool. A huge cluster is continuously running the SAGE engine.
- 1/3rd of Windows 7 security were bugs found by SAGE!

Recent and open-source DSE tools:

- SymCC: compiles program with build-in DSE
- SYMSAN: based on Data-Floow Sanitizer (DFSan)
- Driller: augmenting AFL with symbolic execution

# Discussion

# Symbolic execution is slowly getting more practical

- 1976: A system to generate test data and symbolically execute programs (Lori Clarke)
- 1976: Symbolic execution and program testing (James King)
- 2005-present: practical symbolic execution
  - Moore's Law
  - Better theorem provers (SAT / SMT solvers)
  - Heuristics to control exponential path explosion
  - Improved heap and environment modeling techniques
  - …

# Smart fuzzers vs symbolic: why not both?

Winner of DARPA's Cyber Grand Challenge (CGC)

› Goal was to automatically find and exploit vulnerabilities

› They **combine both** (see presentations from Shellphish)

## American Fuzzy Lop + angr

SHELLPHISH

AFL

- state-of-the-art instrumented fuzzer
- path uniqueness tracking
- genetic mutations
- open source

angr

- binary analysis platform
- implements symbolic execution engine
- works on binary code
- available on github

# Example: the sendmail crackaddr Bug

- Discovered 2003 by Mark Dowd Buffer: overflow in an email address parsing function of Sendmail. Consists of a parsing loop using a state machine (~500 LOC).
- **Bounty for Static Analyzers** since 2011 by Halvar Flake: Halvar extracted a smaller version of the bug as an example of a hard problem for static analyzers (~50 LOC).
- **Found automatically in CGC by ShellPhish** via smart fuzzing and symbolic execution (driller and angr).

Sources:
- http://2015.hackitoergosum.org/slides/HES2015-10-29%20Cracking%20Sendmail%20crackaddr.pdf
- https://thefengs.com/wuchang/courses/cs492/Slides/07_Fuzzing_SymbolicExecution.pptx

# Practical use cases of symbolic execution

- Analysis of course code
- Assure safety / analysis of code
- Reserve engineering and deobfuscation
  - Deobfuscation: recovering an OLLVM-protected program
  - Miasm: free and open-source reverse engineering framework
- Many more…

# Summary

Symbolic execution is a bug finding technique based on automated theorem proving:

- Evaluates the program on **symbolic inputs, and a solver finds concrete values** for those inputs that lead to errors.
- Many success stories in the open-source community and industry.
- Can produce concrete test cases. But cannot, in general, prove the absence of errors