



CS451 – Software Analysis

Lecture 15

Symbolic Execution

Elias Athanasopoulos
elathan@cs.ucy.ac.cy

Overview



- Symbolic execution tracks metadata about a program's state
 - Exactly as we do with taint analysis
- In taint analysis we can reason if specific data that is processed in taint sinks comes from taint sources
- In symbolic execution we can reason about which specific input can drive a program to a specific state
- As it holds with taint analysis, symbolic execution can happen at the source or binary level

Applications



- Creation of inputs that reach specific code states
- Increase code coverage
 - In software testing, or fuzzing if we test for security vulnerabilities, we need to artificially create inputs
 - These inputs should exercise as much of the analyzed code as possible
- Programs that are dynamically analyzed
 - Computing the right payload is not trivial
- Symbolic execution is powerful, but can face scalability issues, when the program's size is increased

Symbolic vs concrete execution



- When we run or analyze a program, we use concrete values
 - These values taken from the input initialize and set variables of the program, again, with concrete values
 - The memory of the process is filled in with specific data
- In symbolic execution, the program is *emulated* with symbolic values instead of concrete ones
- Symbolic execution at the binary level implies that certain memory cells or h/w registers contain symbolic information

Symbolic state



- Symbolic execution replaces concrete values with symbols ($\alpha_i, i \in \mathbb{N}$) that represent a range of concrete values
- The symbolic execution engine constantly computes
 - A set of symbolic expressions
 - A set of path constraints

Symbolic expression



- A symbolic expression φ_j , with $i \in \mathbb{N}$, corresponds either to a symbolic value, α_i , or to some mathematical combination of symbolic expressions, such as: $\varphi_3 = \varphi_1 + \varphi_2$
- The symbolic execution engine maintains a store, σ , with all symbolic expressions

Path constraint



- The path constraint encodes the limitations imposed on the symbolic expressions by the branches taken during execution
- For example, if the symbolic execution takes a branch if $(x < 5)$ and x is mapped to φ_1 , then we have a path constraint: $\varphi_1 < 5$
- All path constraints are stored in π



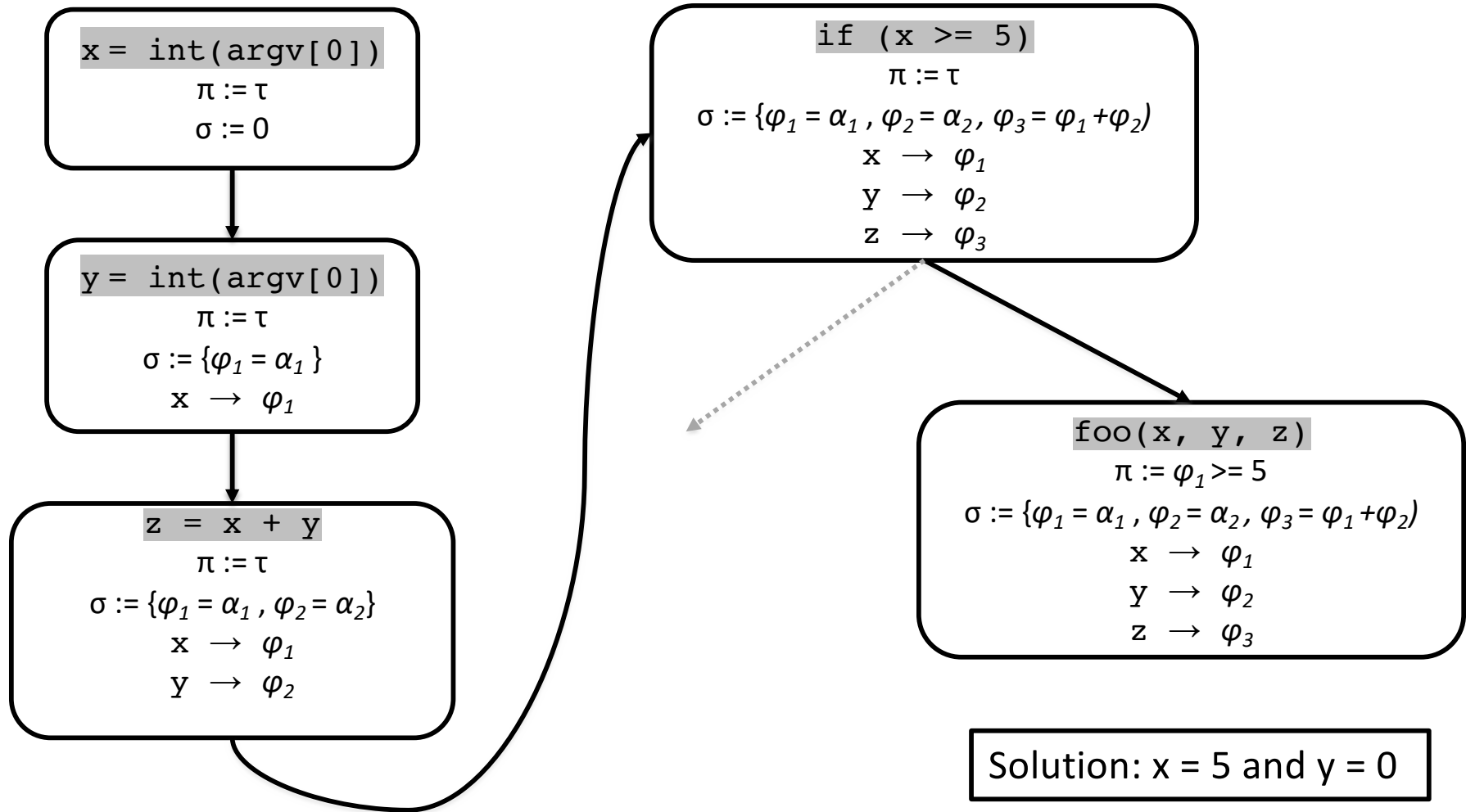
Example

```
x = int(argv[0])  
y = int(argv[1])  
z = x + y
```

```
if (x >= 5)  
    foo(x, y, z)  
    y = y + z  
    if (y < x)  
        baz(x, y, z)  
    else  
        qux(x, y, z)  
else  
    bar(x, y, z)
```

What kind of inputs do we need to reach the call of foo() or bar()?

Executing the program symbolically



Variants and limitations



- There are different types of symbolic execution engines, which can be used for building analysis tools and other applications
- Static vs dynamic
 - The engine may emulate the program statically or actually executing it dynamically (concolic execution)
- Online vs offline
 - The engine may explore multiple paths in parallel or not
- Symbolic state
 - Which parts of the program are represented symbolically and which are not
- Path coverage
 - Which (and how many) program paths the engine explores

Static symbolic execution



- Symbolic execution can be performed statically by emulating all branches symbolically
- Advantages
 - The analysis can be applied on any code, even on code that runs on a different architecture
- Disadvantages
 - Hard to emulate all branches and to emulate parts outside of your control (kernel, third-party library)
 - *Effect modelling* tries to model the behavior of a part you do not control, but it is hard in practice (you need to model the network, the filesystem, etc.)
 - *Direct external interactions* may actually perform the call, but again if multiple calls need to be performed the case becomes complex

Dynamic symbolic execution



- Dynamic symbolic execution or *concolic execution* runs the program with concrete values but keeps a symbolic state
 - Symbolic state is tracked using metadata, as we do with taint analysis
- Does not explore multiple paths in parallel, but only a single path with a concrete value
 - To explore different paths, it flips path constraints and uses the constraint solver to compute concrete inputs that lead to an alternative branch
- Much scalable compared to static symbolic execution
 - No need to maintain state for parallel paths
 - Supports external interactions
- Code coverage is based on concrete values and may be low

Online vs offline



- Online symbolic execution explores multiple paths in parallel, while offline explores only a given path
 - Usually, static symbolic execution is online and dynamic symbolic execution is offline, but there are variants
- Online has the advantage of not running the same code multiple times, however the needed symbolic state can be significant

Symbolic state



- Many frameworks allow to define which parts of the memory is going to be treated as symbolic and which as concrete
 - This approach is more scalable and the constraints can be much easier to solve
- Some engines make memory accessing also symbolic
 - Fully symbolic memory attempts to model all the outcomes of a memory load/store operation (e.g., if you read from an array $a[i]$, where you know i is unsigned and $i < 5$, you are going to read all elements $a[0]..a[4]$)
 - Address concretization attempts to put concrete bounds in cases of unbound memory accessing



Path coverage

- Exploring *all* possible paths can lead to the *path explosion problem*
- Focus on specific paths using heuristics
 - A bug finding tool may focus on sensitive parts, such as loops indexing buffers
- Use a DFS approach to explore each path deeply before moving to another one
 - Sometimes problems arise in deeply nested code
- Concolic execution explores one path at a time
 - You need to restart the program with a different input to explore more paths
 - You can use snapshots for avoiding a complete restart

Increasing scalability



- Simplifying constraints
 - Limiting the number of symbolic variables, by selecting the interesting ones (not an easy problem, sometimes taint analysis can help)
 - Limiting the number of symbolic operations (e.g., if you are interested in an indirect branch that involves `%rax`, then you can execute symbolically only the operations that contribute to the value of `%rax`)
 - Simplifying symbolic memory
- Avoiding the constraint solver