Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

Symbolic Execution

February 4, 2022

Patrick Schönberger

Design Informatik Medien
Hochschule RheinMain

CONTENTS

INTRODUCTION

WHAT IS SYMBOLIC EXECUTION?

- → Method of proving properties of a program
- → Execute the program, assigning symbolic values to variables
- → Prove for all possible inputs
- → Fork execution at control flow statements

SMT SOLVERS

→ Satisfiability Modulo Theories (SMT) solvers determine whether a combination of constraints can be satisfied

→ If yes, they provide concrete values which satisfy the constraints

→ The values on which the constraints can act are determined by the underlying theory

→ Constraints have to be encoded as logical formulas

EXAMPLE PROBLEM

- → Alice is either twice as old as Bob or 2 years younger than Bob.
- → Bob is either twice as old as Alice or 3 years younger than Alice.
- → Alice's age is more than 0 and is not 2

- → $a = 2 * b$ *OR* $a = b - 2$
- → $b = 2 * a$ *OR* $b = a - 3$
- → $a < 0$ *AND NOT* $a = 2$

Taken from [2, p. 14]

EXECUTION MODELS

- → Executing everything symbolically provides complete coverage, but is not always feasible
- → Some side effects might not be possible to provide symbolically
- → Symbolic execution can be mixed with concrete execution (concolic execution)

DYNAMIC SYMBOLIC EXECUTION

→ Program runs symbolically and concretely at the same time

→ Concrete execution guides symbolic execution

→ Paths don’t need to be verified, since they were chosen by the concrete execution

SELECTIVE SYMBOLIC EXECUTION

- → Explore *interesting* sections of code symbolically and the rest concretely
- → Changes in execution mode happen at function level
- → When switching from symbolic to concrete execution, simply concretize arguments
- → When switching from concrete to symbolic execution, turn arguments into symbolic values
- → After symbolic exploration is done, also run concretely for return value

SYMBOLIC BACKWARD EXECUTION

- → Start execution at a specified end point
- → Backtrack to the entry point
- → Useful when trying to find code paths to a specific line of code

PATH SELECTION

→ Symbolic Execution Engines can run multiple paths in parallel

→ Paths can be checked and the most promising ones can be run first

→ Depending on system constraints, paths are usually selected based on memory usage or code coverage

PRIORITIZE MEMORY USAGE

- → When memory is limited, Depth-first search is the most common option
- → Explore each path to the end
- → Maximum memory usage is reached at the longest individual path

PRIORITIZE CODE COVERAGE

→ The most common option to maximize code coverage is Breadth-first search (BFS)

→ Explores every possible branch before continuing any one of them further

→ Instead of simply executing one branch after the other, some methods apply heuristics to choose the path that will provide the most code coverage

PRIORITIZE CODE OF INTEREST

- → Heuristics can also be applied to choose code paths that are relevant to the analysis being done
- → A common approach is to select paths containing loops or memory accesses, since these are known to contain more errors like buffer overflows

CHALLENGES

MEMORY MODEL

- → Memory access introduces a level of indirection into the execution engine
- → The execution engine has to maintain a memory model which maps memory addresses to symbolic values
- → Problems arise when memory addresses contain symbolic values

FULLY SYMBOLIC MEMORY

- → The most obvious solution is to treat all memory addresses as fully symbolic
- → This accurately models memory and can support all memory operations.
- → Only usable if the number of possible addresses is limited

ADDRESS CONCRETIZATION

- → To avoid overstraining the solver, addresses can be concretized to a single value before invoking it
- → This way code paths might be missed
- → A combination of both approaches is known as Partial Memory Mapping
- → Addresses which are written to are always concretized
- → Addresses which are read from are modelled symbolically, if the range of possible values is within a specified threshold

ENVIRONMENT INTERACTION

- → When calling code outside of the analysed codebase, symbolic values leave and enter the execution engine
- → One option is to concretize the arguments and actually call the underlying system
- → Side effects can influence each other when executing paths in parallel
- → Needed system services can be modelled by execution engine
- → Alternatively, the system can be virtualized, with one instance per fork

PATH EXPLOSION

- → A new execution is forked every time a conditional branch is encountered
- → To reduce the number of paths to explore, one can ignore paths that are irrelevant to the analysis or reuse results from previous runs
- → Invoking the solver at every branch ensures unsatisfiable paths are never taken

FUNCTION AND LOOP SUMMARIZATION

- → Functions and loops usually produce the most forks
- → They can be summarized by detecting dependencies between the inputs and the return value/variables in the loop body
- → Loops can generally only be summarized when they are not nested and contain no additional control flow

PATH EQUIVALENCE

→ In large code bases, identical code paths are often found multiple times

→ Identifying these allows ignoring redundant paths and reusing cached results to speed up exploration of new paths

CONSTRAINT SOLVING

- → SMT solvers are still a bottleneck for symbolic execution engines
- → Prominent approaches to optimize this are to reduce the complexity of constraints and reducing calls to the solver
- → Constraints can often be statically optimized without changing their semantics, similar to how an optimizing compiler changes a program without affecting its functionality
- → Dividing constraints into sub-constraints allows for more effective caching, and can also be used to find redundant sub-constraints that can be removed

BINARY VERIFICATION

- → Alternative to symbolic execution for verifying code
- → Symbolic execution can also be used to analyse binary code, with the added benefit of having an SMT solver to prove specific constraints
- → But the usual limitations still apply

- → The problem described in [4], verifying Absence of Runtime Errors (ARTE) and Absence of Privilege Escalation (APE), requires little to no annotation using their method
- → To solve the same problem with a symbolic execution engine, the constraints would have to be provided manually
- → Because symbolic execution is not specific to this problem, using the method described in [4] would probably always be the better alternative
- → In general it has some advantages, most notably being able to model many different properties via logical formulas

[1] Roberto Baldoni et al. "A Survey of Symbolic Execution Techniques". In: *ACM Comput. Surv.* 51.3 (May 2018). ISSN: 0360-0300. DOI: 10.1145/3182657. URL: https://doi.org/10.1145/3182657.

[2] Clark Barrett, Daniel Kroening, and Thomas Melham. *Problem solving for the 21st century: Efficient solver for satisfiability modulo theories*. English (US). Knowledge Transfer Report, Technical Report 3. London Mathematical Society, Smith Institute for Industrial Mathematics, and System Engineering, June 2014.

[3] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "The S2E Platform: Design, Implementation, and Applications". In: *ACM Trans. Comput. Syst.* 30.1 (Feb. 2012). ISSN: 0734-2071. DOI: 10.1145/2110356.2110358. URL: https://doi.org/10.1145/2110356.2110358.

[4] Olivier Nicole et al. "No crash, no exploit: automated verification of embedded kernels". In: *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2021, pp. 27–39.