Symbolic Execution

P. Schönberger

RheinMain University of Applied Sciences, Wiesbaden, Germany

1. Introduction

1.1. What is Symbolic Execution?

Symbolic Execution is a method of proving certain properties of a program. This is done by executing the program in a symbolic execution engine. When the engine executes code, values are represented as symbols and expressions containing these symbols, instead of concrete values. That way, properties can be proven for all possible input values, without explicitly trying all of the, potentially infinite, possibilities. When the symbolic execution engine hits control flow statements, the execution is forked. Each fork has it's own separate symbolic value store and a logical formula that contains the information about every control flow path that was taken in order to arrive at that fork.

1.2. SMT Solvers

SMT (Satisfiability Modulo Theories) solvers are used to determine whether a combination of constraints can be satisfied or not. The types of values these constraints can contain depend on the underlying theory. A simple example would be the theory of real arithmetic, meaning the solver can reason about constraints containing real numbers and operations on them. Other theories allow reasoning about Boolean logic, arrays and matrices, strings, linked lists, trees, etc. On top of that, available theories can be freely combined, making it possible to include for example lists of numbers without any explicit support for them.

SMT solvers take their inputs as questions of the form "given some conditions *C*, is it possible for *X* to happen, and if so, how?", where *C* and *X* are logical formulas. If it finds the constraints to be satisfiable, it also produces values for the constraints that satisfy them. Internally, the supported theories are translated into Boolean formulas, utilizing an SAT solver to check their satisfiability. SAT was the first problem to be proven to be NP-complete, but in practice many constraints can be solved, and very efficiently too.

In order to answer a question, it has to be encoded as logical formulas. The tools available for this encoding usually include basic arithmetic, uninterpreted functions and bit-vectors to model digital data. Some SMT solvers also provide quantifiers, which are useful but increase complexity by quite a bit.

2. Execution Models

Running all of the code in the execution engine symbolically allows for complete coverage, and can prove properties for all possible code paths. In reality however, especially for larger code bases, it is often unfeasible to do so because the number of paths to take grows exponentially. Also, depending on the environment the code runs in, the execution engine might not be able to simulate all the necessary side effects (see 4.2). To account for this, symbolic execution is often mixed with concrete execution, where parts of the program are executed with actual values obtained from their previous symbolic ones. This is referred to as concolic execution and typically trades soundness for performance.

Different approaches are taken to mix symbolic and concrete execution, the most prevalent of which are presented below.

2.1. Dynamic Symbolic Execution

Dynamic Symbolic Execution (DSE) works by running the program symbolically and concretely at the same time. The concrete execution guides the symbolic execution; everytime a branch is taken, the symbolic execution is directed towards that same branch. This has the added benefit of mitigating the need to verify the path, since the concrete execution proves it can be taken.

2.2. Selective Symbolic Execution

Instead of the concrete execution driving the process, in Selective Symbolic Execution (SSE) both methods are mixed. Only portions of the code that are of interest to the analysis are explored symbolically, while the majority is explored concretely. The switch between symbolic and concrete execution happens on function calls. When going from symbolic to conrete execution, the arguments provided by the symbolic execution are concretized, the function is evaluated concretely, and then execution resumes symbolically. When going from concrete to symbolic execution, the concrete arguments are made symbolic and the function is explored fully symbolically. It is also explored concretely, and the result of the concrete execution is returned.

2.3. Symbolic Backward Execution

Symbolic Backward Execution (SBE) is a variant of Symbolic Execution, where execution starts at a specified end point in the code and backtracks to the entry point. When encountering a branch condition, it is explored symbolically, like with forward symbolic execution, and paths which are not satisfiable are discarded, since they cannot lead to the specified end point. This is useful when trying to find code paths that lead to a specific line of code, for example when debugging.

3. Path Selection

Since symbolic execution allows for exploration of many paths in parallel, the computational cost of exploring all of them can potentially be very high. To maximize effectiveness, paths are checked beforehand in an attempt to execute the most promising ones first. Depending on the system constraints, the approaches to select paths typically prioritize either memory usage or code coverage, since they are intrinsically linked.

3.1. Prioritize Memory Usage

Depth-first search (DFS) is the most common and a very simple option when working with memory constraints. DFS simply explores a path to the end and then continues to the next one. This way the tree of explored paths grows linearly, basically only having one long active branch at a time.

3.2. Prioritize Code Coverage

Optimizing code coverage is a little harder, since it is not always obvious how much code any given path will contain. The simplest and most common method here is breadth-first search (BFS), which starts exploring every possible path at every branch condition. This means that many different paths are discovered quickly, but since they are all executed in parallel, exploring them fully takes a lot of memory and time. Instead of simply executing one path after the other, some methods apply heuristics to determine which path will lead to the largest gain in code coverage.

3.3. Prioritize Code of Interest

Analogously to code coverage, heuristics are also applied to determine code paths that might be more relevant for analysis. A simple approach to this is to pick paths which have contained small errors before, assuming that these paths have not been properly tested and might contain more errors. Similar approaches favor paths in which loops or memory accesses have been identified, based on the observation that errors like buffer overflows happen there most often.

4. Challenges

4.1. Memory Model

In addition to directly storing variables, most programming languages allow using pointers and arrays. These introduce a level of indirection which has to be dealt with by the symbolic engine. The solution is some memory model that supports associating memory addresses with expressions over both concrete and symbolic values. This also allows supporting regular variables like before, by simply referring to them by their address instead of their name. Problems arise as soon as an address includes symbolic values and there are various solutions to this.

4.1.1. Fully Symbolic Memory

The simplest strategy is to treat all memory addresses as fully symbolic. Being the most general approach, this can accurately model memory and supports all memory operations. This strategy is usable if the number of possible memory addresses is limited, otherwise the number of possible states can quickly become way too large.

4.1.2. Address Concretization

In order to avoid this large number of states when dealing with unbounded memory addresses, symbolic addresses can be concretized to a single value before invoking the solver. The tradeoff is that paths that depend on specific values for some pointers might be missed.

A combination of these methods is known as Partial Memory Modeling. Addresses which are being written to are always concretized and addresses which are read from are modelled symbolically, if the range of possible values is below a certain threshold. This approach is a compromise both in terms of complexity and effectiveness.

4.2. Environment Interaction

Most programs utilize code outside of their own codebase. Examples of this are calls to the operating system, calling outside of a managed runtime, like calling native functions from within the JVM, or any closed source components. In all of these cases, symbolic values leave and enter the analysed code.

The obvious solution is to just concretize the arguments and actually call the underlying system. But not only does concretizing values reduce the number of explored paths, since paths are often executed in parallel, these calls can interfere with each other, for example when two paths access the same file at the same time.

This can be overcome by modelling the system services. Symbolic engines exist that can model file systems, thread and process organization, sockets and environment variables. But these models are expensive to create and maintain, and still only provide limited functionality. That is why an alternative approach for operating system services is to interact with the real system in a virtualized environment, such that different explored paths cannot interfere with one another. Every time a branch condition is evaluated and a path is forked, a new instance of the virtualized system is spawned. Because this quickly becomes very expensive, path selection has to be limited (see 3).

4.3. Path Explosion

Symbolic execution engines typically fork off new paths to explore every time a conditional branch is encountered. This can quickly lead to an exponentially growing number of paths, which in turn directly impacts the runtime and memory costs. The main ways to avoid this is to ignore paths that are irrelevant to the analysis being done, and to reuse results from previous analyses.

A very simple possibility is to invoke the constraint solver at every branch. This way unrealizable paths are never explored. Since it is very simple to implement and does not limit generality, this is the default behaviour of most symbolic execution engines. Taking this a step further, some executors reduce found unsatisfiable conditions as far as possible while still maintaining unsatisfiability, so that any future condition which contains it can immediately be discarded.

4.3.1. Function and Loop Summarization

The main culprit of path explosion are loops and functions. Since they are traversed multiple times, it is possible to create a summary of their execution to allow for easier analysis on future invocations. Function summaries can be dynamically created by overlapping constraints discovered on the input and output values during exploration.

Loops are generally summarized by detecting dependencies between the loop condition and symbolic values in the loop body. This allows to avoid executions of the loop both in the same program state and under different conditions. Loops can generally only be summarized when they are not nested and do not contain any other conditional branches.

4.3.2. Path Equivalence

In a large enough state space, there are usually many redundant paths that can be abstracted in order to avoid paths that cannot provide new information to the current analysis. Furthermore, paths that share the same semantics do not need to be repeated if they have been executed before. When analysing programs which are marked with explicit error locations, the executor can check at every branch whether the current path is subsumed by any path that previously failed to reach an error location. This only works reliably under the condition that all possible routes through a path which is declared to be unable to reach some error location have been explored. Choosing the right path selection strategy can therefore be very important. DFS specifically is a good fit here, since paths are explored in their entirety before moving on to the next one. Alternatively, Greedy Confirmation can be used with other path selection strategies, where additional traversals are performed for nodes whose paths have not been fully explored.

4.4. Constraint Solving

Even though SMT solvers have become increasingly efficient in recent years, which made their use in symbolic execution feasible in the first place, their performance and the limits of what they can do still provide a bottleneck for symbolic execution engines. In order to tackle this, prominent approaches are reducing the complexity of constraints, reducing the number of invocations of the SMT solver and finding alternatives for constraints that solvers inherently struggle with.

Reducing how often the solver is called, and also how complex the constraints it is called with are, is a straight forward way to improve performance. Similar to how a compiler can statically optimize portions of a program, constraints can often be simplified without changing their semantics. Furthermore, constraints can be divided into subconstraints, which not only synergizes well with caching the results of previous runs, it can also remove redundant constraints that don't actually impact the result.

When caching constraint results, on top of skipping constraints that have already been tested for satisfiability, constraints which contain any cached and unsatisfiable constraints can immediately be declared unsatisfiable aswell. Similarly, if a constraint is contained in another one that was previously proven to be satisfiable, it can also be declared satisfiable.

5. Binary Verification

Binary verification of code, as in Binsec/Codex, presents a recent alternative to symbolic execution. Symbolic execution can also be used to analyse binary code, and using an SMT solver provides the usual benefit of being able to express conditions intuitively. But the known limitations still apply, especially being unable to properly handle unbounded loops, leading to path explosion.

The problems arising from memory modelling in symbolic execution engines don't really apply to binary verification, at least not in the same way. Memory is modelled, but since there are no memory addresses containing symbolic values, they cannot lead to an explosion in states.

Since execution is not forked at conditional instructions, path explosion does not apply either.

The given problem, verifying the Absence of Runtime Errors (ARTE) and Absence of Privilege Escalation (APE) in a kernel, requires little to no annotation when using Binsec/Codex. Also, since no solver is used, the cost of running it can be omitted. A symbolic execution engine would need annotations to describe the constraints, since it is not specific to this task. On the other hand this means that symbolic execution can be used in a wider range of applications, since it is more general in nature. As long as the constraints are expressable as logical formulas, symbolic execution can be used to check them.

So for the specific task of proving APE and ARTE for open and closed source kernels, binary verification does indeed seem like the preferable solution. For analysing code in general however, symbolic execution has it's own advantages, most notably being able to model many different properties via logical formulas which can then be checked automatically.

References

[BCD*18] Baldoni R., Coppa E., D'Elia D. C., Demetrescu C., Finocchi I.: A survey of symbolic execution techniques. *ACM Comput. Surv. 51*, 3 (may 2018). URL: `https://doi.org/10.1145/3182657`, `doi:10.1145/3182657`.

[BKM14] Barrett C., Kroening D., Melham T.: *Problem solving for the 21st century: Efficient solver for satisfiability modulo theories*. Knowledge Transfer Report, Technical Report 3. London Mathematical Society and Smith Institute for Industrial Mathematics and System Engineering, June 2014.

[CKC12] Chipounov V., Kuznetsov V., Candea G.: The s2e platform: Design, implementation, and applications. *ACM Trans. Comput. Syst. 30*, 1 (feb 2012). URL: `https://doi.org/10.1145/2110356.2110358`, `doi:10.1145/2110356.2110358`.

[NLBR21] NICOLE O., LEMERRE M., BARDIN S., RIVAL X.: No crash, no exploit: automated verification of embedded kernels. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2021), IEEE, pp. 27–39.