1 goto fail: A Framing Example

Consider Apple’s goto fail bug in a part of their SSL code. A repeated `goto fail;` statement prevented a crucial call to `sslRawVerify()`, so the code would always exit with success without actually completing the main verification. This poses all kinds of security issues. A couple of possible ways to check for this kind of bug:

- We might require some mutations to data are made before flow control is altered
- A more general solution might be to require that the crucial function in question is always called along all paths

2 Symbolic Analysis

The aforementioned approach can be further generalized by asking: given a proposition about a program, is there some input for which this proposition is satisfied? This question is answered by analyzing the program symbolically, hence the term “symbolic analysis.” It is worth briefly and generally differentiating this approach from static analysis:

- Branches are not merged here but rather global dependencies are tracked. This cuts down on the number of false positives, which occur at a higher rate in static analysis
- Symbolic analysis cannot keep track of memory side effects (due to an exponential growth in the inputs)

2.1 Relation to Boolean SAT

The italicized question above bears a clear resemblance to the boolean satisfiability problem, which asks: given a logical proposition in conjunctive normal form, is there a setting of literals such that the proposition is satisfied? In fact, solving symbolic analysis problems is done via a similar process.

2.2 Satisfiability Modulo Theories

SMTs are decision problems of the form posed above. Namely, they consist of a property of a program and the question of whether some inputs satisfy that property. They can be formulated essentially as a SAT problem with some “syntactic sugar,” namely theories that describe integers, real numbers, arrays, lists, etc. This concept is easily illustrated through a simple example:
\[ b+2=c \text{ and } f(\text{read(write}(a,b,3), c-2)) != f(c-b+1) \]
\[ \Rightarrow b+2=c \text{ and } f(\text{read(write}(a,b,3), b+2-2)) != f(b+2-b+1) \text{ by substitution} \]
\[ \Rightarrow b+2=c \text{ and } f(\text{read(write}(a,b,3), b)) != f(3) \text{ by arithmetic} \]
\[ \Rightarrow b+2=c \text{ and } f(3) != f(3) \text{ by array theory axiom} \]
\[ \Rightarrow \text{Not Satisfiable (assuming } f \text{ is deterministic)} \]

In general, boolean SAT and SMT are NP hard problems, but in practice large classes of problems are not too difficult to resolve.

### 2.3 Automatic Test Generation

We might take symbolic analysis a step further and ask specifically which inputs satisfy the proposition in question. The intuition behind the approach to this problem is:

- Divide input space into equivalence classes (the idea being that all inputs in a given class will induce the same execution path)
- Test one example from each equivalence class

This notion of equivalence classes of inputs helps to achieve the goal of having good path coverage.

### 2.4 Symbolic Execution

Symbolic execution consists of executing a program with symbolic-valued inputs. Again, the goal is to cover as many paths as possible. We can represent equivalence classes of inputs via first-order logical formulas describing path constraints. This allows for the checking of the validity of a given branch.

### 2.5 Practical Issues

Some problems with symbolic execution:

- **Loops and Recursion** lead to infinite execution trees
- **Path explosion** (i.e. there are exponentially-many paths)
- **Solver problem** (SMT solvers cannot resolve all path constraints)
- **Coverage problem** (Might not be possible to reach deeper parts of the tree)

### 2.6 Concolic Execution

A solution to the coverage problem is concolic execution. “Concolic” is a portmanteau of “concrete” and “symbolic.” The idea is to reach deeper into the execution tree by simultaneously executing on concrete and symbolic inputs.