Abstract

Software is rapidly becoming a key component for an increasing number of critical applications ranging from automotive, aerospace, banking, healthcare, and many more. Formal software verification, which involves verifying the software properties for all possible inputs, has thus become crucial. But the absence of the formal guarantees for the compiler that is used to translate the software to the final executable code limits the guarantees provided by the formal source code verification.

Equivalence checking involves verifying the functional equivalence between a program specification and its implementation. An equivalence checker can be used in a translation validation approach to provide formal correctness guarantees for the executable code generated by the compiler. A black-box equivalence checker takes a program-pair as input and makes minimal assumptions on the exact nature of transformations performed from one program to another.

The general equivalence checking problem is undecidable and is very challenging in the translation validation context due to the potentially large syntactic gap between the source and assembly representation and the long and complex nature of transformations/optimizations performed by the modern optimizing compilers. These optimizations result in significant structural differences

between the input source code and the optimized output code. *This work proposes two algorithms that help make significant progress in the space of robust translation validation.*

The first algorithm efficiently finds the correlation between program transitions for structurally significantly different (but bisimilar) program-pairs and the second algorithm finds a general class of relations between state elements of programs that have significant syntactic gap across them. Both these algorithms are static and do not rely on execution traces. Instead, they purely rely on the concrete models (aka counterexamples) returned by the SMT solvers.

The third contribution of this work is *the first black-box equivalence checking tool that can automatically compute equivalence across the unoptimized intermediate representation (IR) of a program and its optimized x86 assembly implementation generated either by an optimizing compiler or developed by a human programmer.* We use a custom IR representation that resembles LLVM IR to specify the input program. The long and rich pipeline of transformations from the unoptimized IR to the optimized x86 assembly includes optimizations like loop unrolling, peeling, unswitching, versioning, loop inversion, vectorization, register allocation, code hoisting, strength reduction, dead code elimination, and many more.