CMPT 479: Team Symbad

Team Members

Cole Greer

Oscar Smith-Sieger

The Problem

Why in 2018 is it still a job to spend 100 hours running into a wall at every possible angle? We really don't know, but we think it's a bit of a problem. Our idea is to utilize automated software analysis techniques to attempt to find and diagnose problematic program states in highly state-based programs. In our case, this is games.

Existing Work

The game development industry is generally way behind the times when it comes to modern software development techniques. In searching, we were unable to find any comprehensive test systems in use for game development. In general, all that is done by most existing systems is simple static analysis culminating in little more than linting.

An example of a tool ahead of the times (but still from 2012!) is "WallMonster"^1 used in the development of *The Witness*. This tool is still fundamentally rudimentary and very case-specific. It is quite hard to extend the actual tool (though not the process) to other games or engines.

Our Idea

Our attempt to solve this problem is based around performing static analysis, in a similar form to symbolic execution, to reason about the state of individual objects within the program.

The concept is based around identifying individual state-based objects within the program, such as the player, enemies, etc. These objects are considered state-based since the game as a whole operates on a tick-based system, providing a sort of "time" which controls all objects.

Once identified, our model is to determine whatever variables, in a symbolic fashion, can affect the state of the object in a given tick. This would form a "function" of sorts which takes these variables as input, and provides the updated object state as output.

By identifying all these factors which can change an object, we can construct a dependency graph of information between different objects in a given tick. Then we can take this information and determine the domain of the inputs for these objects. From there, we can, by simple union of all these domains, determine the set of all inputs to which the program responds.

Fundamentally, the model of tracking the program state via the symbolic object state is the most useful part of the project. This is because that information can then be extrapolated to provide the basis of many other analysis tools.

The Tools and Platform

We plan to attempt to implement this concept for the Unity game engine. This is due to the fact it is free, and well-used within the industry. Unity primarily uses C# for its programming, which means we're unable to use LLVM for our analysis. In its stead, we found a tool called Mono.Cecil which provides at least some of the features of LLVM. It will allow us to investigate and modify the compiled library files used in C# and the .NET platform.

Evaluation

To determine the effectiveness of our system we plan to use a very simple metric: does it produce useful output. That may seem simplistic, but fundamentally the core goal of ours is to create the base model which can later be extended to provide more useful core analysis.

Stretch Goals

If we're able to complete the core system to produce the model, we have a number of extensions we've thought of which would use the information to provide actual benefit.

The first of these would be a module allowing a user to place a constraint on the domain of a given variable within an object. From there, the change would propogate through the model producing a new set of possible inputs. The compliment of this new set taken in the universe of the set of all inputs the program responds to (aka, the un-constrained inputs) would be the set of inputs producing the unintended outcome. A developer can then use this information to ensure the condition is being enforced, through the inputs maintaining their constraint.

Another concept would be to use the dependency information to discover the data dependencies and non-dependencies through the program. This would provide a basis for a system to identify parallelizable sections of the program.