Отчёт

Начальный набор входных данных

SMT-LIB2 достаточно сложный язык для того, чтобы можно было реализовать мутацию написанной на нем формулы в общем виде, поэтому я решил работать лишь с созданным мной форматом, являющимся подмножеством этого языка.

Сам формат включает в себя нисколько секций:

1. Определения операций
2. Определение символьных переменных
3. Определение дерева вычислений
4. Ограничение на корень дерева

Каждой секции соответсвует определенный лейбл, выраженный в программе как “; label_name”. Каждая секция начинается с этого лейбла.

Дерево вычислений может использовать в себе определенные в 1-ой секции функции и сивольные переменный, а также константы.

Ограничение наложенное на корень дерева – это предикат, решатель в процессе своей работы ищит интерпритацию при условии истинности данного предиката.

Естественно любая строчка формулы должна соответсвовать определенному, весьма строгому синтаксису, что позволяет реализовать максимально простой десериализатор.

Сам начальный набор является практически полностью синтетическим, т.е. сгенерированным мутатором, но с той лишь только разницей, что вероятность мутации была значительно выше, чем при фаззинге. Для того чтобы его получить, была описана теорема Деморгана, а также в секции 1 были описаны порядка 25 операций, не все они используются в этой формуле, но могут возникнуть в результате мутации. К формуле с теоремой Деморгана последовательно применялась мутация, после чего практически случайным образом были выбраны формулы, попавшие в итоговый начальный набор. Я посчитал это приемлимым, так как утилита afl-showmap показала, что они значительно отличаются друг от друга своим покрытием. Из-за большой вероятности мутации

полученные формулы не имеют ничего общего с исходной формулой, по сути ее можно рассматривать как seed генерации. В итоге был получен набор сильно различающихся формул, к которым в процессе фаззинга будут применятся не такие сильные мутации, как в процессе генерации.

Естественно было бы лучше получить начальный набор естественных тестов, но приведение произвольной формулы к формату, который бы подходил для мутации – это весьма трудоемкое занятие, а добовление в начальный набор произвольных формул я посчитал нецелесообразным, так как к ним не может быть применена мутация. Поэтому я решил, что будет гораздо целесообразней сфокусироваться на мутаторе и создать автоматическое получение достаточно разнообразных формул.

В итоге полученный набора начальных данных является разнообразным и корректным. В разнообразности можно убедиться, посмотрев директорию base.

Кастомный мутатор

Для кастомной мутации были реализованы десериализатор и сериализатор для созданного формата. Сама мутация применялась к дереву, получаемому в результате десериализации 3-ей секции. По сути это дерево является графом вычислений.

Применяемые мутации:

1. Произвольная перестановка аргументов
2. Замена операции на константу или символьную переменную
3. Замена константы или символьной переменной на операцию
4. Замена одной операции на другую
5. Замена константы на другую константу или символьную переменную
6. Тоже что и 5, только для символьных переменных

Во время мутации используются операции и символьные переменные, определенные в 1-ой и 2-ой секциях входной формулы.

Корректность мутатора напрямую зависит от корректности сериализатора, десериализатора и проводимых с узлами графа мутаций. Проводилось лишь ручное тестирование, но в результате него не было обнаружено ошибок, z3 на формулах, полученных в результате мутации,

отрабатывал произвольное количество времени, при этом с почти равной вероятностью получался как sat, так и unsat результат. При этом проверялось, что результат работы мутатора не только является корректной формулой, но и заново может быть передан в мутатор.

Данных мутаций достаточно для того, чтобы со временем, в результате последовательного приминения мутатора, получить произвольное выражение.

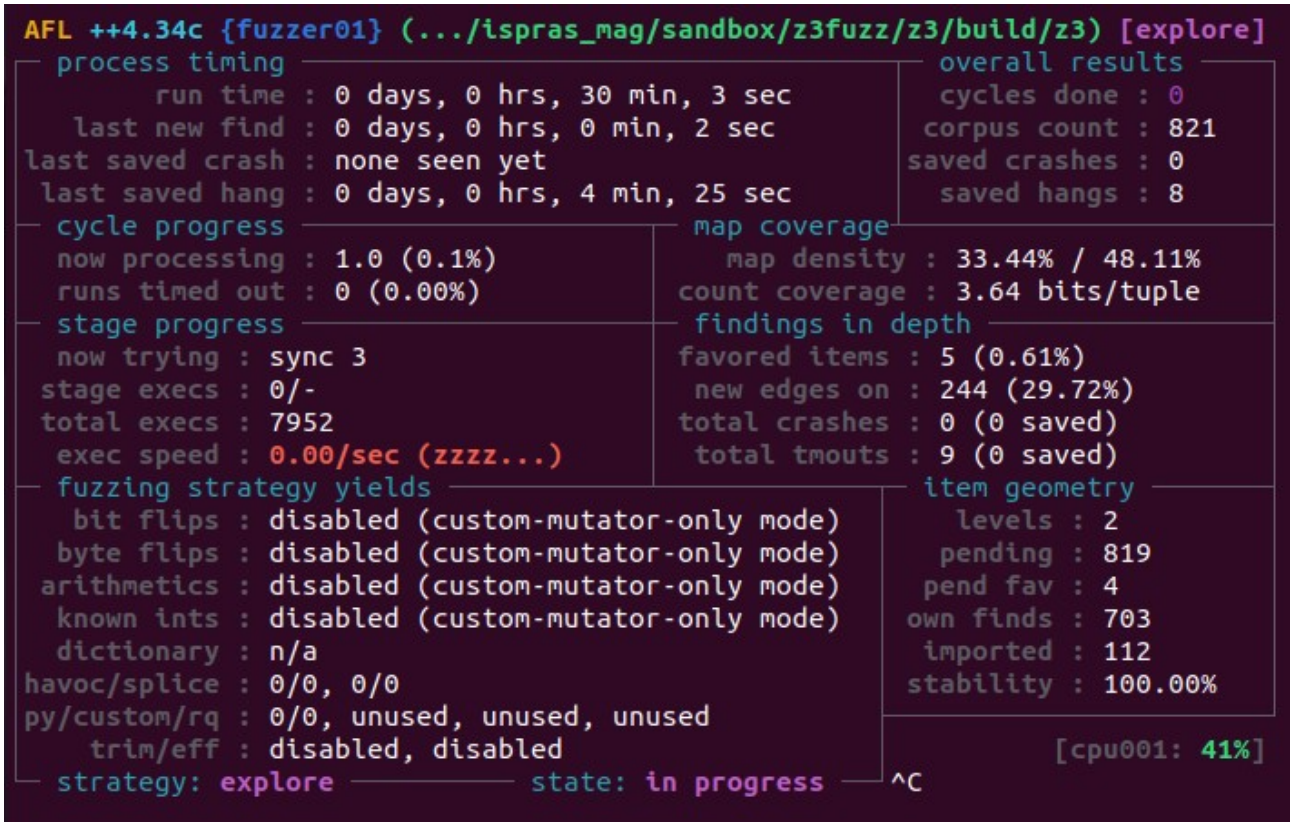
На что нацелен фаззинг

Я выбрал теорию битовых векторов, т.к. в ней больше операций, чем в других теориях. Но с одним ограничением, а именно любая функция, константа или символьная переменная должны иметь тип bv64, что было введено с целью упрощения графа вычислений и реализации производимых мутаций.

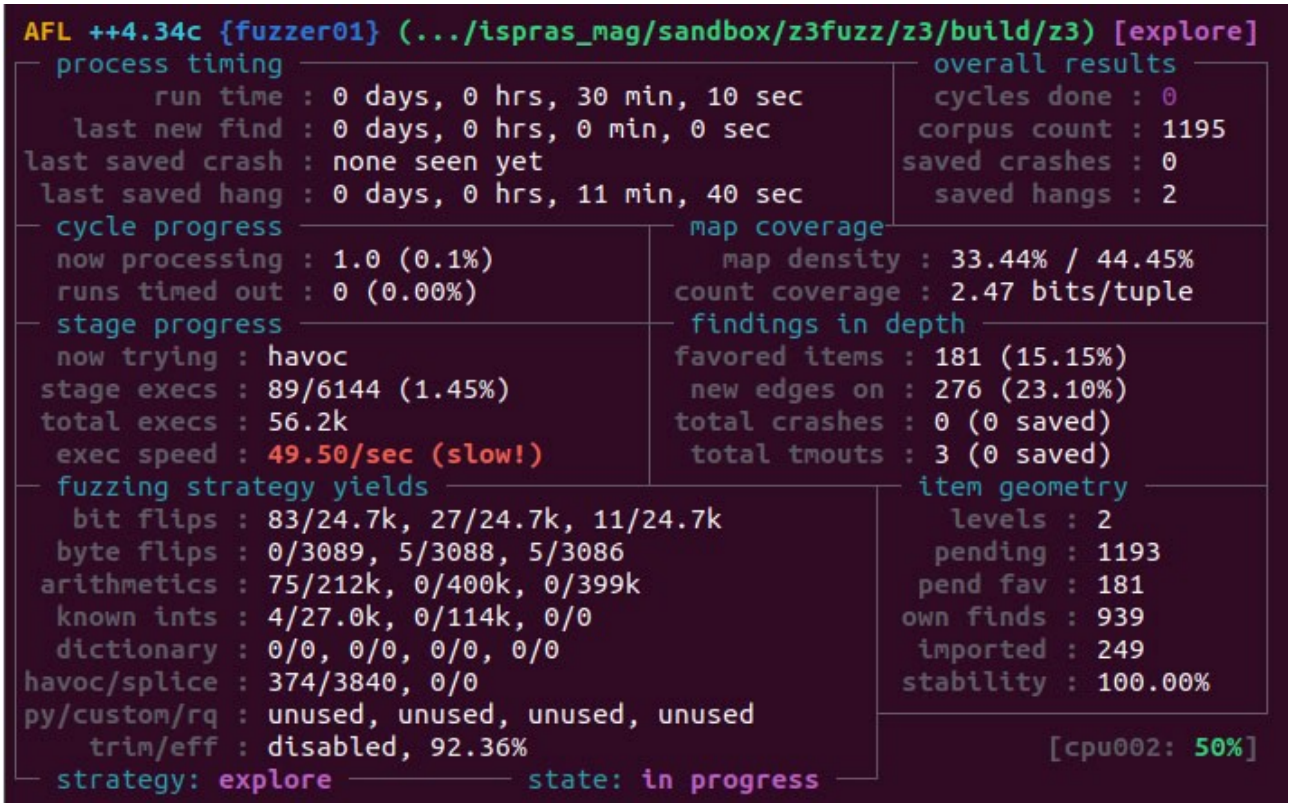
По сути проверяется то, как в решателе реализована поддержка битовых векторов и операций.

Результат запуска

Кастомный мутатор



Без кастомного мутатора

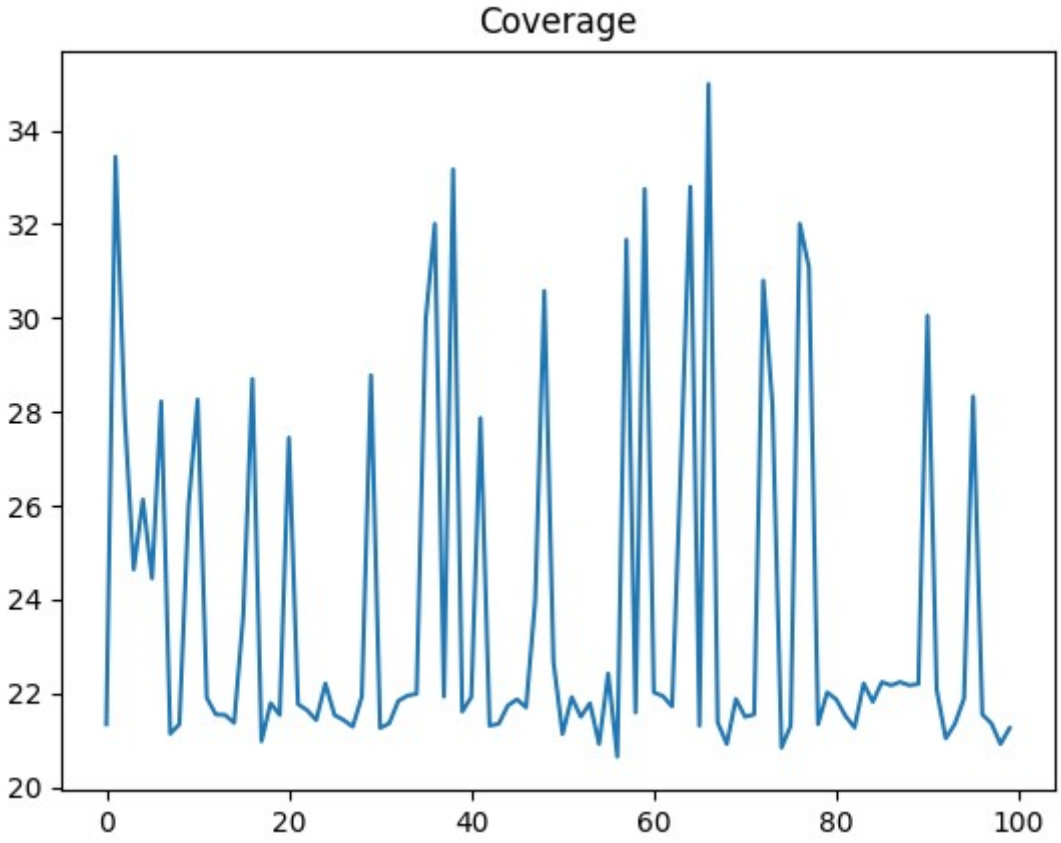


Начальный набор дает 39% покрытия, соответсвенно за 30 минут запуска кастомный мутатор дает 9,1% покрытия, а встроенные в afl++ 5,5%. Но их нельзя сравнивать напрямую, так как кастомный мутатор генерует валидные формулы, а встроенные, как я понял, производят практически рандомные мутации и генерируют формулу, которая немного отличается от исходный, но эквивалентна с точки зрения солвера, либо генеруют невалидные входные данные, это подтверждается запуском солвера на входных данных, сгенерированных встроенными мутаторами. Соответственно встроенные мутаторы фаззят то, как солвер обрабатывает некорректные входные данные, а кастомный мутатор генерирует только корректные, соответсвенно они покрывают разные компоненты солвера.

График покрытия от входных данных

В качестве последовательность входных данных была взята очередь главного процесса, полученная при фаззинге.

Покрытие снималось при помощи afl-showmap.


Coverage
34
32
30
28
26
24
22
20
0
20
40
60
80
100