Automating fuzz-testing for C projects

Daniel Wait daniel.wait22@imperial.ac.uk mentored by Paul Butcher butcher@adacore.com and João Azevedo azevedo@adacore.com

Background

C is a very popular language in high-integrity software. Yet, the language is very bug-prone if used incorrectly

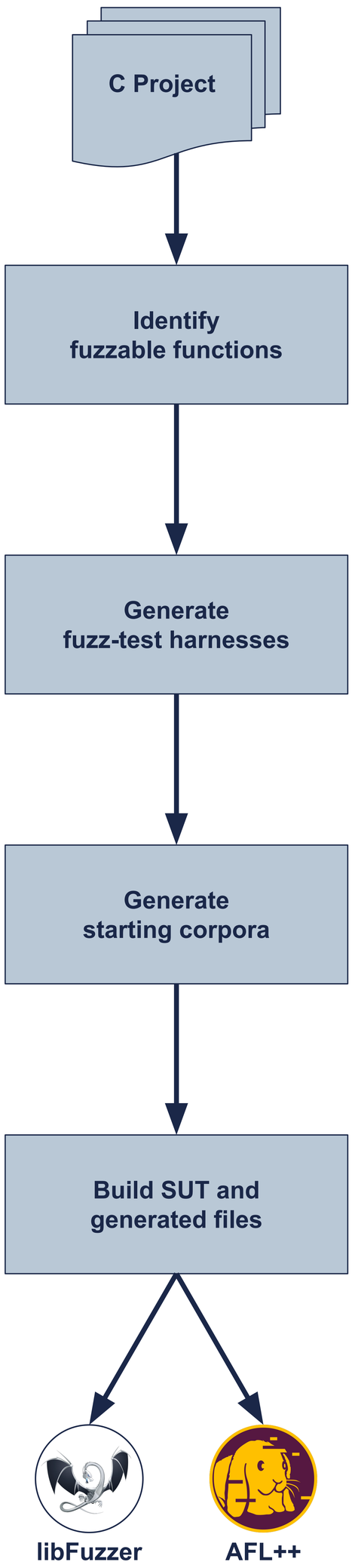
Fuzz-testing is one way to discover vulnerabilities and unhandled failures. Although, the technique is difficult and requires significant effort to set it up effectively since each fuzz-target in an SUT must be manually set up

Aim

To automate the process of fuzz-testing a C project's public API

Why?
- Reduce the skill barrier and time/labour cost of setting up fuzzing
- Increase scalability and code coverage
- Provide consistent and reliable results by avoiding human error

Workflow



Language Ambiguity

- C has ambiguous syntax
- Pointers and arrays cannot be distinguished

- Associating arrays and lengths is just a convention
- Conventions cannot be inferred from the syntax

- All this must be declared for effective fuzzing

my_code.h

```
// array pointer with its length
int sum_array(int *arr, int n);

// unrelated pointer and value
int set_reg(int *reg, int x);
```

Generic Marshalling

- The fuzz-engine provides a binary blob to the fuzz-test harness, but the SUT is a C function so it needs structured parameters passed ("marshalled") to it

- A schema is generated for each SUT to model its parameters for a (de)serialisation library

- This schema can also be used to validate the provided binary before deserialisation to ensure crashes only happen in the SUT and not in the fuzz-test harness

- A high degree of generality was required to support C's type system so a templating system with "recipes" for each type is used to allow for aggregate types and recursive types

Starting Corpus Generation

- A good starting corpus is small, valid, and diverse in order to maximise code coverage

- Without statically analysing the body of each SUT, this is a difficult problem

- The automated starting corpus generation considers every combination of "interesting" initial value for each parameter to the SUT

- "Interesting" meant boundaries values (e.g. `INT_MIN`, `INT_MAX`), special values (e.g., `NaN`, Infinity), and a few random values

The Test Oracle Problem

- Unlike unit testing, which checks for expected behaviour, fuzz testing checks for no unexpected behaviour

- There is no oracle for the lack of unexpected behaviour so the crash oracle is used instead
- For C, however, the lack of runtime checks leaves the crash oracle unsuitable as many instances of undefined behaviour do not cause the program to crash

- Instrumenting the SUT with code sanitizers (ASan, MSan, UBSan, etc.) strengthens our test oracle
- Some common vulnerabilities now detectable by the stronger oracle are buffer overflows and use-after-frees via ASan, uninitialised variables via MSan, and null-pointer dereferences via UBSan