Fuzzing Boot Camp

BY: VRIG

AGENDA

Date

```c
#include <stdio.h>
#include <string.h>

int main(){
    char input[100];
    fgets(input, 100, stdin);
    if (input[0] == '1'){
        if (input[1] == '2'){
            if (input[2] == '3'){
                if (input[3] == '4'){
                    if (input[4] == '5'){
                        printf("CASE 5\n");
                    }
                }
                else{
                    printf("CASE 3\n");
                }
            }
            else{
                printf("CASE 4\n");
            }
        }
    }
}
```

01 What is the point of fuzzing?

- Find the “impossible”
- Crash code

02 American Fuzzy Lop

- Biggest innovation in low level bug funding in recent memory
- Coverage Analysis as fuzzing metric

03 What is Coverage Analysis ?

- Code Depth
- Useful for finding unexpected code paths

American fuzzy lop

The first coverage guided fuzzer, insane step forward in the world of security research came out in 2013

LibFuzzer

The LLVM based coverage guided fuzzer that is the backbone of the OSS-Fuzz system very useful for getting information on your harness and code coverage via fuzz-introspector

AFLplusplus

The current easy plugin state of the art universal fuzzer. Its mostly commonly used for setting up fast fuzzing campaigns where the complexity is in the harness and not the mutation strategy

HonggFuzzer

A feedback-driven, instrumentation-based grey-box fuzzer. It supports both software and hardware coverage, works well in many general fuzzing tasks.

LibAFL

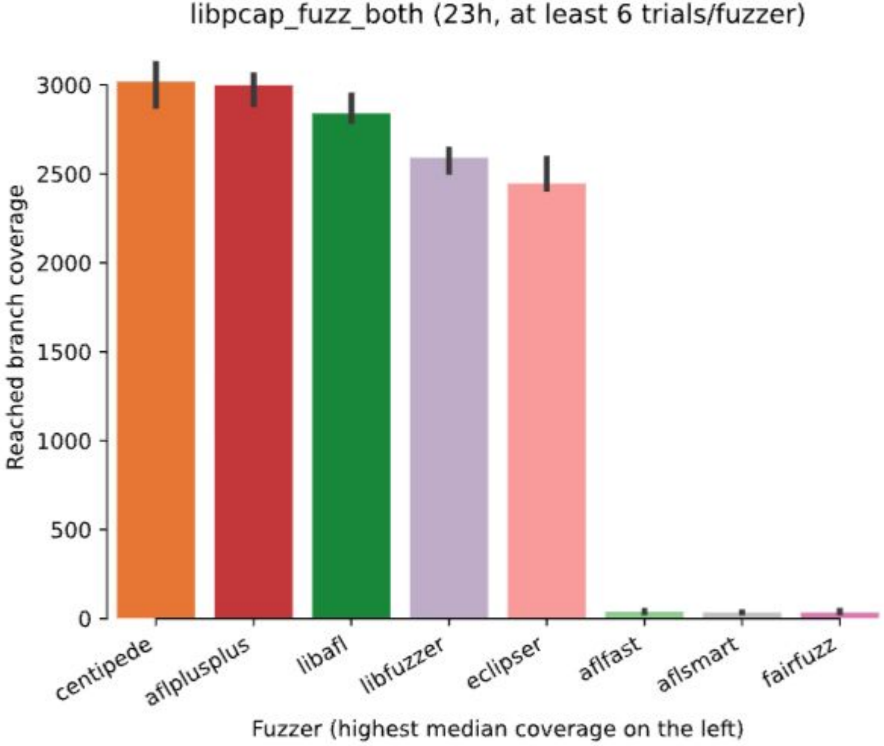
A rust based fuzzer library that allows for creation of complex fuzzers. Created by the same team as AFL++, but with better performance and the ability of a library

Centipede

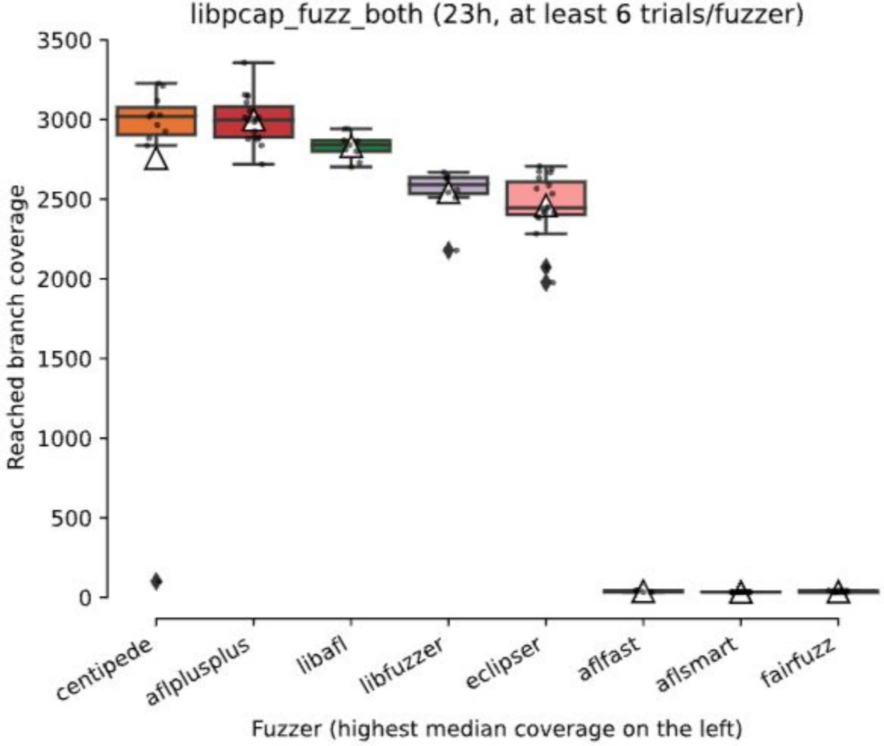
Google research project based on the idea of creating a fuzzer for large complex code bases that allows for easy integration into LLVM sanitization and has modular feedback metrics

libpcap_fuzz_both summary

Ranking by median reached code coverage

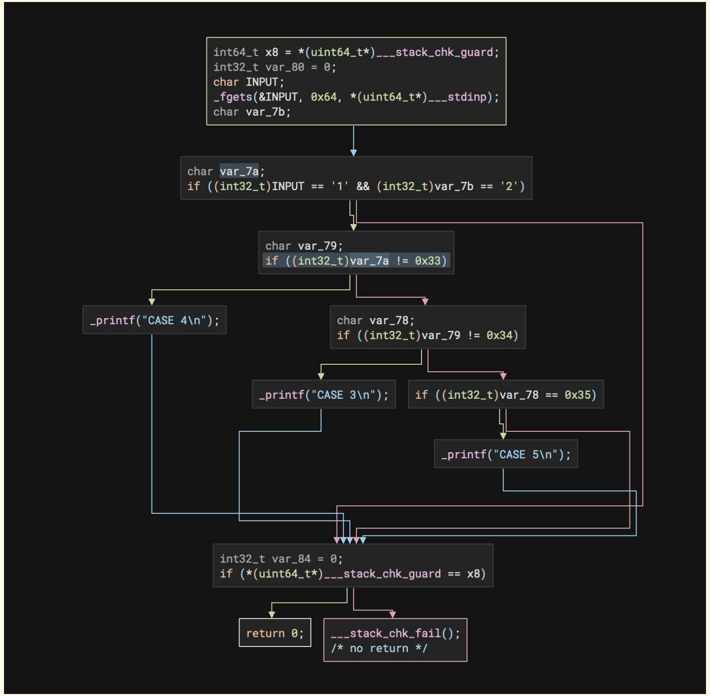


Reached code coverage distribution



Date

Coverage Guided Mutation - Standard Fuzzing



Save_if_interesting

→ calculate_new_bits_if_necessary

→ has_new_bits_unclassified

→ has_new_bits(afl, virgin_map);

→ discover_word

Key Points

- There are different mutation strategies
  - They will result in changes generally at Save_if_interesting

{"explore", "mmopt", "exploit", "fast", "coe", "lin", "quad", "rare", "seek"}

- Fuzz level: the number of fuzzing iterations
  - When you do a full fuzzing iteration this inc the value in “havoc_stage"

This is in the afl-fuzz-bitmap

Fuzzing Modes

Fuzzing via standard function

- afl-fuzz -i inputs -o findings -- ./target_program @@
    - This is the mosi basic format of fuzzing where you are just feed in raw data

Forked-Server:

Fuzzing via instrumented binaries but still via stdin

- Modern fuzzers require you to build your biniares with instruments so they can keep track of coverage. This enables our coverage guided fuzzing.
    - afl-clang-lto/afl-clang-lto++ | clang/clang++ 11
    - afl-clang-fast/afl-clang-fast++ | clang/clang++ 3.8

Persistent mode:

- Our fuzzer now no longer is sending data via IO, instead we create a fuzzing “harness” which allows our fuzzer to directly send data to the program or library in question allowing for much much faster fuzzing.
    - A Harness is one of the core components of proper high level fuzzing

LibFuzzer Fuzzing Entry Point

```
#include <stdio.h>
#include <stdint.h>


int LLVMFuzzerInitialize(int *argc, char ***argv) {
  return 0;
}

int LLVMFuzzerTestOneInput(const uint8_t *data,
size_t size) {
  return 0;
}
```

AFL++ Fuzzing Entry Point

```
int main(int argc, char **argv)
{
   (void)argc; (void)argv;

   ssize_t len;
   unsigned char *buf;

   __AFL_INIT();
   buf = __AFL_FUZZ_TESTCASE_BUF;
   while (__AFL_LOOP(INT_MAX)) {
       len = __AFL_FUZZ_TESTCASE_LEN;
       LLVMFuzzerTestOneInput(buf, (size_t)len);
   }

   return 0;
}
```

Writing Good Harness:

1. The first rule of any good fuzzing harness is to identify good entry points
    a. At the top of code flow, use CGF's
    b. Where raw data / unstructured information is passed in
        i. You want to allow the fuzzers to find interesting code paths
2. Keep your harness targeted
    a. Fuzzing a single group of code paths with optimized harnessing is better than a universal harness
        i. Don't use the same fuzzing harness for multiple code paths
3. Make sure to clean up your state and values
    a. Avoid memory leaks and causing memory corruption in your own harness
        i. This can result in tons of false positives
4. Make sure you are using the LLVMFuzzerInitialize to avoid heavy and tedious initialization loops that aren't required for fuzzing per execution

Writing Good Harness 2:

1. Don’t allow for a piece of the fuzzing data to be used as multiple parts of the data follow
2. Make sure to fully reset state when possible while fuzzing for each execution
3. Remove all debug in both your harness and source code
4. Do initial input validation
    a. If the data is too small
        i. If the data is going to result in total useless arbitrary input that will immediately fail create useful filters or generate it in a meaningful way in the harness or via mutations
        ii. Good seeds also fix this issues
5. When fuzzing make sure to edit source code to optimize for fuzzing environment
    a. If you are fuzzing parsing logic but for some reason there is a math or slow hashing algorithm consider avoiding or stubbing it
    b. Stub out or create Mocks of any non-important subroutines / interconnected systems

Fuzzer Flags & Building

Different instrumentation levels:

1. afl-clang-lto/afl-clang-lto++
   a. Fastest and works by compiling at link time to minimize edge collisions
2. afl-clang-fast/afl-clang-fast++
3. afl-gcc-fast/afl-g++-fast

Fuzzer Environment flags:

- export AFL_USE_ASAN=1
  - Address SANitizer, finds memory corruption vulnerabilities like use-after-free, NULL pointer dereference, buffer overruns, etc
- export AFL_USE_MSAN=1
  - Memory SANitizer, finds read accesses to uninitialized memory, e.g., a local variable that is defined and read before it is even set
- export AFL_USE_UBSAN=1
  - Undefined Behavior SANitizer, finds instances where - by the C and C++ standards - undefined behavior happens, e.g., adding two signed integers where the result is larger than what a signed integer can hold.
- export AFL_USE_CFISAN=1
  - Control Flow Integrity SANitizer, finds instances where the control flow is found to be illegal.In fuzzing, this is mostly reduced to detecting type confusion vulnerabilities - which is, however, one of the most important and dangerous C++ memory corruption classes!

Fuzzer Flags & Building 2

Fuzzer Environment flags:

- AFL_USE_TSAN=1
  - TSAN = Thread SANitizer, finds thread race conditions. Enabled with export AFL_USE_TSAN=1 before compiling.
- AFL_USE_LSAN=1
  - LSAN = Leak SANitizer, finds memory leaks in a program. This is not really a security issue, but for developers this can be very valuable. Note that unlike the other sanitizers above this needs __AFL_LEAK_CHECK(); added to all areas of the target source code where you find a leak check necessary! Enabled with export AFL_USE_LSAN=1 before compiling. To ignore the memory-leaking check for certain allocations, __AFL_LSAN_OFF(); can be used before memory is allocated, and __AFL_LSAN_ON(); afterwards. Memory allocated between these two macros will not be checked for memory leaks.

Libfuzzer should have these enabled by default, with AFL you have to pick certain systems to run with certain flags since not all flags can be used at once on

- For example: ASAN and MSAN can't be used with each other
- ASAN and CFISAN can't be used with each other

∗Building with environment flags tends to slow down the fuzzer dramatically depending on the target and flag you should run a spread out mix of them

Input compiler avoidance tag:

```
#ifdef FUZZING_BUILD_MODE_UNSAFE_FOR_PRODUCTION
  // say that the checksum or HMAC was fine - or whatever is required
  // to eliminate the need for the fuzzer to guess the right checksum
  return 0;
#endif
```

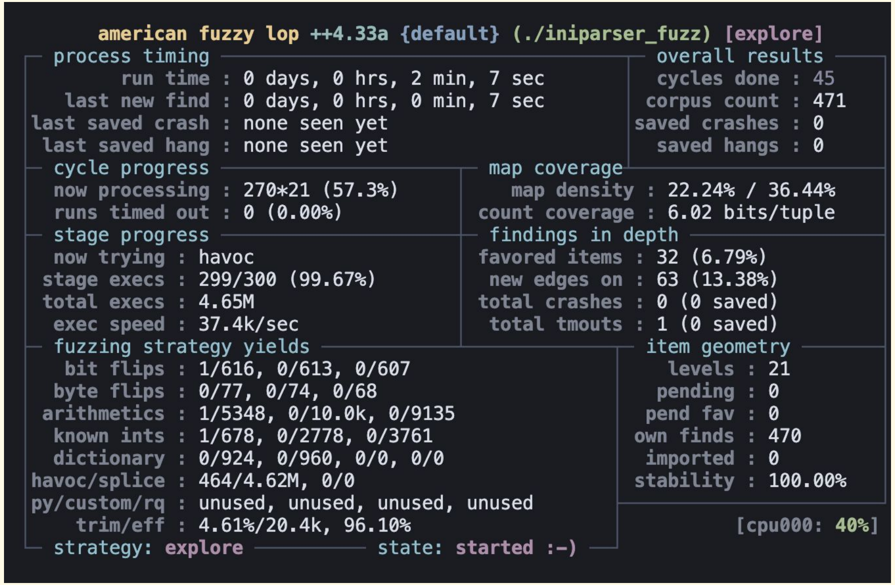
Installing AFL++ & Quick Demo

https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/INSTALL.md

- Go here and install

https://github.com/Dudcom/fuzzing-vrig-slides

```
docker pull aflplusplus/aflplusplus:latest
docker run -ti -v /location/of/your/target:/src aflplusplus/aflplusplus

Mkdir in
Mkdir out

AFL_USE_ASAN=1 afl-clang-lto -g -O1 -fsanitize=address,undefined \
iniparser_fuzz.c iniparser.c dictionary.c -o iniparser_fuzz

Afl-fuzz -i in -o out iniparser_fuzz
```



Installing LibFuzzer & Quick Demo

If you are on the afl docker make sure to update

Comment out lines 93-125

apt install clang llvm

clang -DNO_MAIN -g -O2 -fsanitize=fuzzer iniparser_fuzz.c iniparser.c dictionary.c -o iniparser_fuzz

```
#686969 NEW    cov: 182 ft: 1094 corp: 469/66Kb lim: 2094 exec/s: 68696 rss: 837Mb L: 215/1817 MS: 5 CopyPart-CMP-PersAutoDict-EraseBytes-InsertRepeatedB
E: "'''"-"\267\246\337E"-
#687217 REDUCE cov: 182 ft: 1094 corp: 469/66Kb lim: 2094 exec/s: 68721 rss: 837Mb L: 201/1817 MS: 3 ShuffleBytes-InsertByte-EraseBytes-
#687643 REDUCE cov: 182 ft: 1094 corp: 469/66Kb lim: 2094 exec/s: 68764 rss: 838Mb L: 174/1817 MS: 1 EraseBytes-
#687780 REDUCE cov: 182 ft: 1094 corp: 469/66Kb lim: 2094 exec/s: 68778 rss: 838Mb L: 828/1817 MS: 2 ChangeByte-EraseBytes-
#688001 REDUCE cov: 182 ft: 1094 corp: 469/66Kb lim: 2094 exec/s: 68800 rss: 838Mb L: 147/1817 MS: 1 EraseBytes-
#688297 REDUCE cov: 182 ft: 1094 corp: 469/66Kb lim: 2094 exec/s: 68829 rss: 839Mb L: 72/1817 MS: 1 EraseBytes-
#688493 REDUCE cov: 182 ft: 1094 corp: 469/66Kb lim: 2094 exec/s: 68849 rss: 839Mb L: 33/1817 MS: 1 EraseBytes-
#689088 REDUCE cov: 182 ft: 1094 corp: 469/66Kb lim: 2094 exec/s: 68908 rss: 840Mb L: 120/1817 MS: 5 ShuffleBytes-CopyPart-EraseBytes-EraseBytes-CopyPart
#689734 REDUCE cov: 182 ft: 1094 corp: 469/66Kb lim: 2094 exec/s: 68973 rss: 840Mb L: 107/1817 MS: 1 EraseBytes-
#690022 REDUCE cov: 182 ft: 1094 corp: 469/66Kb lim: 2094 exec/s: 69002 rss: 841Mb L: 66/1817 MS: 3 ChangeBit-ChangeASCIIInt-EraseBytes-
#690690 REDUCE cov: 182 ft: 1094 corp: 469/66Kb lim: 2094 exec/s: 69069 rss: 842Mb L: 70/1817 MS: 3 ShuffleBytes-ShuffleBytes-EraseBytes-
#690801 REDUCE cov: 182 ft: 1094 corp: 469/66Kb lim: 2094 exec/s: 69080 rss: 842Mb L: 30/1817 MS: 1 EraseBytes-
#691280 REDUCE cov: 183 ft: 1095 corp: 470/67Kb lim: 2094 exec/s: 69128 rss: 842Mb L: 1320/1817 MS: 4 PersAutoDict-CopyPart-InsertByte-CopyPart- DE: "\37
77\377\377\377\377\023"-
#693003 REDUCE cov: 183 ft: 1095 corp: 470/67Kb lim: 2105 exec/s: 69300 rss: 844Mb L: 1462/1817 MS: 3 ChangeBit-ShuffleBytes-EraseBytes-
#694105 REDUCE cov: 183 ft: 1095 corp: 470/67Kb lim: 2105 exec/s: 69410 rss: 845Mb L: 541/1817 MS: 2 ChangeBit-EraseBytes-
#694647 REDUCE cov: 183 ft: 1095 corp: 470/67Kb lim: 2105 exec/s: 69464 rss: 846Mb L: 1666/1817 MS: 2 ChangeASCIIInt-EraseBytes-
#695428 REDUCE cov: 183 ft: 1095 corp: 470/67Kb lim: 2105 exec/s: 69542 rss: 847Mb L: 183/1817 MS: 1 EraseBytes-
#695534 REDUCE cov: 183 ft: 1095 corp: 470/67Kb lim: 2105 exec/s: 69553 rss: 847Mb L: 601/1817 MS: 1 EraseBytes-
#695897 REDUCE cov: 183 ft: 1095 corp: 470/67Kb lim: 2105 exec/s: 69589 rss: 847Mb L: 147/1817 MS: 3 ShuffleBytes-PersAutoDict-EraseBytes- DE: "\203\203"
#697381 REDUCE cov: 183 ft: 1095 corp: 470/67Kb lim: 2116 exec/s: 69738 rss: 849Mb L: 575/1817 MS: 4 ChangeBit-ShuffleBytes-CopyPart-EraseBytes-
#699092 REDUCE cov: 183 ft: 1095 corp: 470/67Kb lim: 2127 exec/s: 69909 rss: 851Mb L: 573/1817 MS: 1 EraseBytes-
#699183 REDUCE cov: 183 ft: 1095 corp: 470/67Kb lim: 2127 exec/s: 69918 rss: 851Mb L: 48/1817 MS: 1 EraseBytes-
#699266 REDUCE cov: 183 ft: 1095 corp: 470/67Kb lim: 2127 exec/s: 69926 rss: 851Mb L: 1312/1817 MS: 3 CrossOver-InsertRepeatedBytes-EraseBytes-
#700901 REDUCE cov: 183 ft: 1095 corp: 470/67Kb lim: 2138 exec/s: 70090 rss: 853Mb L: 201/1817 MS: 5 CMP-InsertByte-ChangeByte-ShuffleBytes-EraseBytes- D
5\000\000\000"-
#701233 REDUCE cov: 183 ft: 1095 corp: 470/67Kb lim: 2138 exec/s: 70123 rss: 854Mb L: 44/1817 MS: 2 ShuffleBytes-EraseBytes-
#701425 REDUCE cov: 183 ft: 1095 corp: 470/67Kb lim: 2138 exec/s: 70142 rss: 854Mb L: 20/1817 MS: 2 InsertByte-EraseBytes-
```

LibAFL - The Rust Based Fuzzer Library

Created by the same team as AFL++, similar but different

Allows for creation of your own fuzzers, allowing for greater control of mutation strategies and fuzzing methods.

Fuzzers at the end of the day at a very high level are systems to find different states in a programing environment. This means you can further control the feedback system to do things like play and solve games

LibAFL allows its user to create highly modularized fuzzing systems for easy usage, you still need to build your fuzzing harness its just now your fuzzer itself is more powerful.

Breaking down LibAFL

```
use libafl::{
    Error, HasMetadata,
    corpus::{Corpus, InMemoryOnDiskCorpus, OnDiskCorpus},
    events::SimpleRestartingEventManager,
    executors::{ExitKind, ShadowExecutor, inprocess::InProcessExecutor},
    feedback_or,
    feedbacks::{CrashFeedback, MaxMapFeedback},
    fuzzer::{Fuzzer, StdFuzzer},
    inputs::{BytesInput, HasTargetBytes},
    monitors::SimpleMonitor,
    mutators::{
        HavocScheduledMutator, StdMOptMutator, Tokens, havoc_mutations,
        token_mutations::I2SRandReplace, tokens_mutations,
    },
    observers::{CanTrack, HitcountsMapObserver, TimeObserver},
    schedulers::{
        IndexesLenTimeMinimizerScheduler, StdWeightedScheduler,
powersched::PowerSchedule,
    },
    stages::{
        ShadowTracingStage, StdMutationalStage, calibrate::CalibrationStage,
        power::StdPowerMutationalStage,
    },
    state::{HasCorpus, StdState},
};
```

This is the libafl library includes in the rust format

Core parts of the code:

- Error is just error handling
- HasMetadata is a trait that allows for access to metadata maps on the code
- Corpus: where the test inputs are stored-
    - The testcase is the first input
    - The following values are "metadata" on the system describing how the test case is stored ie ondisk or inmemory
    - add() : Add new test case
    - get() : Retrieve test case by ID
    - count() : Get number of test cases
    - current() : Get currently scheduled test case

SimpleRestartingEventManager

- Strut that manager fuzzer events and handles automatic restarts after crashes

Executors::

- Next slide !

Executor

This is the system that will actually execute the code

- If we compare it to other fuzzers like libFuzzer this would be calling the harness function LLVMFuzzerTestOneInput
- The executor is how all “volatile” operations that would be required to run a target once get called

This means that the executor system is responsible for informing the program about the input that the fuzzer wants to use in the run, writing to a memory location for instance or passing it as a parameter to the harness function.

- As such the Executor trait is created to describe parts of the execution process

The trait also uses Observers to connect each of the executions which is why we need the HasOvservers type for any function we define as an executor. You can write your own custom executor or use one of the standard implementations like:

InProcessExecutor

- Execute the harness function inside the fuzzer process
    - Fastest way to execute a harness

ForkserverExecutor

- This allows for the forkserver model with shared memory if needed

InprocessForkExecutor

- This will fork before running the harness that is the only difference between this and InProcessExecutor
    - You do this when the harness is unstable and can destroy global states. Since we are making a child process when using the mode you have to ensure the enviorment can keep track of crashes properly

Breaking down GenericInProcessExecutor

```
pub struct GenericInProcessExecutor<EM, H, HB, HT, I, OT, S, Z> {
       harness_fn: HB, // The function to execute
       inner: GenericInProcessExecutorInner<EM, HT, I, OT, S, Z>, // Core
       state
       phantom: PhantomData<(*const H, HB)>, // Type safety
}
```

The function will take in

- Harness function which is what we actually call on each execution
- GenericInProcessExecutorInner is used to manage and monitor global values and keep track of crashes Allows for pre/post execution hooks as well

```
// From mod.rs lines 85-108
fn run_target(&mut self, fuzzer: &mut Z, state: &mut S, mgr: &mut EM, input: &I)
-> Result<ExitKind, Error> {
    *state.executions_mut() += 1;  // Increment execution counter
    unsafe {
        let executor_ptr = ptr::from_ref(self) as *const c_void;
        self.inner.enter_target(fuzzer, state, mgr, input, executor_ptr);  // Set
global state
    }

    self.inner.hooks.pre_exec_all(state, input);  // Pre-execution hooks
    let ret = self.harness_fn.borrow_mut()(input);  // Call your target function
    self.inner.hooks.post_exec_all(state, input);  // Post-execution hooks
    self.inner.leave_target(fuzzer, state, mgr, input);  // Cleanup global state
    Ok(ret)
}
```

Run_target is used after building the generic to run our system, it calls the enter_target which global states for crash monitoring.

Execution in the lib.rs

```rust
let mut harness = |input: &BytesInput| {
    let target = input.target_bytes();
    let buf = target.as_slice();
    unsafe {
        libfuzzer_test_one_input(buf);  // Your target function
    }
    ExitKind::Ok
};

// Lines 354-361
let executor = InProcessExecutor::with_timeout(
    &mut harness,                 // Your harness function
    tuple_list!(edges_observer, time_observer),  // Observers
    &mut fuzzer,                 // Fuzzer instance
    &mut state,                  // Fuzzer state
    &mut mgr,                    // Event manager
    timeout,                     // Execution timeout
)?;
```

Mutators !

```
use libafl::{
    Error, HasMetadata,
    corpus::{Corpus, InMemoryOnDiskCorpus, OnDiskCorpus},
    events::SimpleRestartingEventManager,
    executors::{ExitKind, ShadowExecutor,
inprocess::InProcessExecutor},
    feedback_or,
    feedbacks::{CrashFeedback, MaxMapFeedback},
    fuzzer::{Fuzzer, StdFuzzer},
    inputs::{BytesInput, HasTargetBytes},
    monitors::SimpleMonitor,
    mutators::{
        HavocScheduledMutator, StdMOptMutator, Tokens,
havoc_mutations,
        token_mutations::I2SRandReplace, tokens_mutations,
    },
    observers::{CanTrack, HitcountsMapObserver, TimeObserver},
    schedulers::{
        IndexesLenTimeMinimizerScheduler, StdWeightedScheduler,
powersched::PowerSchedule,
    },
    stages::{
        ShadowTracingStage, StdMutationalStage,
calibrate::CalibrationStage,
        power::StdPowerMutationalStage,
    },
    state::{HasCorpus, StdState},
};
```

Mutators are what define our mutator strategy. They will take in input and then output a mutated value.

- We can have different types of mutators and are able to define how we use them
- We can create mutation stages as well which allow for further complex or controlled mutations based on the corpus or other metrics of analysis beyond just simple coverage

For example if we wanted to apply a specific json mutation every time we were hitting a very specific error that we implemented as a internal heuristics / call back.

Overall Mutator Path

1. Seed Files
↓
2. InMemoryOnDiskCorpus (memory)
↓
3. Scheduler selects corpus entry
↓
4. Testcase → BytesInput (extract raw data)
↓
5. Mutator.mutate() applies mutations
↓
6. Mutated BytesInput → Executor
↓
7. Executor runs target with mutated input
↓
8. Observers collect coverage/feedback
↓
9. If interesting → Add to corpus
↓
10. Repeat with next corpus entry

Mutators in example code

```
// Lines 388-396 in our lib.rs
if state.must_load_initial_inputs() {
    state.load_initial_inputs(&mut fuzzer, &mut executor, &mut mgr, &[seed_dir.clone()])
        .unwrap_or_else(|_| {
            println!("Failed to load initial corpus at {:?}", &seed_dir);
            process::exit(0);
        });
    println!("We imported {} inputs from disk.", state.corpus().count()); // add rawbytes to the state
object
}

// Line 401
fuzzer.fuzz_loop(&mut stages, &mut executor, &mut state, &mut mgr)?;

// Lines 331-338
let scheduler = IndexesLenTimeMinimizerScheduler::new(
    &edges_observer,
    StdWeightedScheduler::with_schedule(
        &mut state, // pass in the state into the scheduler
        &edges_observer,
        Some(PowerSchedule::fast()),
    ),
);
```

Mutators in example code

```
// Lines 368-369 in our librs
let mut stages = tuple_list!(calibration, tracing, i2s, power);

// Lines 327-328
let power: StdPowerMutationalStage<_, _, BytesInput, _, _, _> =
    StdPowerMutationalStage::new(mutator);
--------------------------------------------------------------------------------
// From LibAFL src code
let num = self.iterations(state)?;  // How many mutations to try
let mut testcase = state.current_testcase_mut()?;  // Get the selected corpus entry

let Ok(input) = I1::try_transform_from(&mut testcase, state) else {
    return Ok(());
};  // extrac the input data from the testcase

for _ in 0..num {  // For each mutation iteration
    let mut input = input.clone();  // clone the input

    let mutated = self.mutator_mut().mutate(state, &mut input)?;  //  MUTATION HAPPENS HERE

    if mutated == MutationResult::Skipped {
        continue;
    }

    // Test the mutated input
    let (untransformed, post) = input.try_transform_into(state)?;
    let (_, corpus_id) = fuzzer.evaluate_filtered(state, executor, manager, &untransformed)?;
}
```

Testing/Demo

git clone https://github.com/AFLplusplus/LibAFL

cargo build --release

https://github.com/AFLplusplus/LibAFL/tree/main/fuzzers/inprocess/fuzzbench

cargo build --release

```
 4/6 (66%)
[Objective #0] run time: 1m-2s, clients: 1, corpus: 2, objectives: 973, executions: 3548, exec/sec: 56.80, stability: 3/3 (100%), edges:
 4/6 (66%)
[Objective #0] run time: 1m-2s, clients: 1, corpus: 2, objectives: 974, executions: 3552, exec/sec: 56.80, stability: 3/3 (100%), edges:
 4/6 (66%)
[Objective #0] run time: 1m-2s, clients: 1, corpus: 2, objectives: 975, executions: 3554, exec/sec: 56.78, stability: 3/3 (100%), edges:
 4/6 (66%)
[Objective #0] run time: 1m-2s, clients: 1, corpus: 2, objectives: 976, executions: 3556, exec/sec: 56.75, stability: 3/3 (100%), edges:
 4/6 (66%)
[Objective #0] run time: 1m-2s, clients: 1, corpus: 2, objectives: 977, executions: 3559, exec/sec: 56.74, stability: 3/3 (100%), edges:
 4/6 (66%)
[Objective #0] run time: 1m-2s, clients: 1, corpus: 2, objectives: 978, executions: 3561, exec/sec: 56.72, stability: 3/3 (100%), edges:
 4/6 (66%)
[Objective #0] run time: 1m-2s, clients: 1, corpus: 2, objectives: 979, executions: 3564, exec/sec: 56.71, stability: 3/3 (100%), edges:
 4/6 (66%)
[Objective #0] run time: 1m-2s, clients: 1, corpus: 2, objectives: 980, executions: 3567, exec/sec: 56.70, stability: 3/3 (100%), edges:
 4/6 (66%)
[Objective #0] run time: 1m-2s, clients: 1, corpus: 2, objectives: 981, executions: 3570, exec/sec: 56.69, stability: 3/3 (100%), edges:
 4/6 (66%)
[Objective #0] run time: 1m-3s, clients: 1, corpus: 2, objectives: 982, executions: 3572, exec/sec: 56.66, stability: 3/3 (100%), edges:
 4/6 (66%)
[Objective #0] run time: 1m-3s, clients: 1, corpus: 2, objectives: 983, executions: 3574, exec/sec: 56.64, stability: 3/3 (100%), edges:
 4/6 (66%)
[Objective #0] run time: 1m-3s, clients: 1, corpus: 2, objectives: 984, executions: 3576, exec/sec: 56.61, stability: 3/3 (100%), edges:
 4/6 (66%)
[Objective #0] run time: 1m-3s, clients: 1, corpus: 2, objectives: 985, executions: 3580, exec/sec: 56.62, stability: 3/3 (100%), edges:
 4/6 (66%)
[Objective #0] run time: 1m-3s, clients: 1, corpus: 2, objectives: 986, executions: 3582, exec/sec: 56.59, stability: 3/3 (100%), edges:
 4/6 (66%)
[Objective #0] run time: 1m-3s, clients: 1, corpus: 2, objectives: 987, executions: 3584, exec/sec: 56.57, stability: 3/3 (100%), edges:
 4/6 (66%)

⌘K to generate a command
```

Grammar/Structure Aware Fuzzing

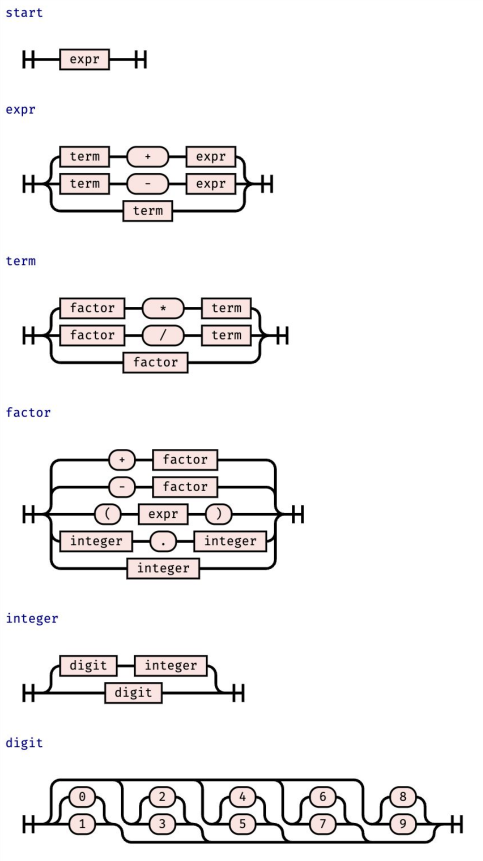
```
simple_grammar_fuzzer(grammar=EXPR_GRAMMAR, max_nonterminals=3, log=True)
```

```
<start> -> <expr>                        <expr>
<expr> -> <term> + <expr>                <term> + <expr>
<term> -> <factor>                       <factor> + <expr>
<factor> -> <integer>                    <integer> + <expr>
<integer> -> <digit>                     <digit> + <expr>
<digit> -> 6                             6 + <expr>
<expr> -> <term> - <expr>                6 + <term> - <expr>
<expr> -> <term>                         6 + <term> - <term>
<term> -> <factor>                       6 + <factor> - <term>
<factor> -> -<factor>                    6 + -<factor> - <term>
<term> -> <factor>                       6 + -<factor> - <factor>
<factor> -> (<expr>)                     6 + -(<expr>) - <factor>
<factor> -> (<expr>)                     6 + -(<expr>) - (<expr>)
<expr> -> <term>                         6 + -(<term>) - (<expr>)
<expr> -> <term>                         6 + -(<term>) - (<term>)
<term> -> <factor>                       6 + -(<factor>) - (<term>)
<factor> -> +<factor>                    6 + -(+<factor>) - (<term>)
<factor> -> +<factor>                    6 + -(++<factor>) - (<term>)
<term> -> <factor>                       6 + -(++<factor>) - (<factor>)
<factor> -> (<expr>)                     6 + -(++(<expr>)) - (<factor>)
<factor> -> <integer>                    6 + -(++(<expr>)) - (<integer>)
<expr> -> <term>                         6 + -(++(<term>)) - (<integer>)
<integer> -> <digit>                     6 + -(++(<term>)) - (<digit>)
<digit> -> 9                             6 + -(++(<term>)) - (9)
<term> -> <factor> * <term>              6 + -(++(<factor> * <term>)) - (9)
<term> -> <factor>                       6 + -(++(<factor> * <factor>)) - (9)
<factor> -> <integer>                    6 + -(++(<integer> * <factor>)) - (9)
<integer> -> <digit>                     6 + -(++(<digit> * <factor>)) - (9)
<digit> -> 2                             6 + -(++(2 * <factor>)) - (9)
<factor> -> +<factor>                    6 + -(++(2 * +<factor>)) - (9)
<factor> -> -<factor>                    6 + -(++(2 * +-<factor>)) - (9)
<factor> -> -<factor>                    6 + -(++(2 * +--<factor>)) - (9)
<factor> -> -<factor>                    6 + -(++(2 * +---<factor>)) - (9)
<factor> -> -<factor>                    6 + -(++(2 * +----<factor>)) - (9)
<factor> -> <integer>.<integer>          6 + -(++(2 * +----<integer>.<integer>)) - (9)
<integer> -> <digit>                     6 + -(++(2 * +----<digit>.<integer>)) - (9)
<integer> -> <digit>                     6 + -(++(2 * +----<digit>.<digit>)) - (9)
<digit> -> 1                             6 + -(++(2 * +----1.<digit>)) - (9)
<digit> -> 7                             6 + -(++(2 * +----1.7)) - (9)

'6 + -(++(2 * +----1.7)) - (9)'
```

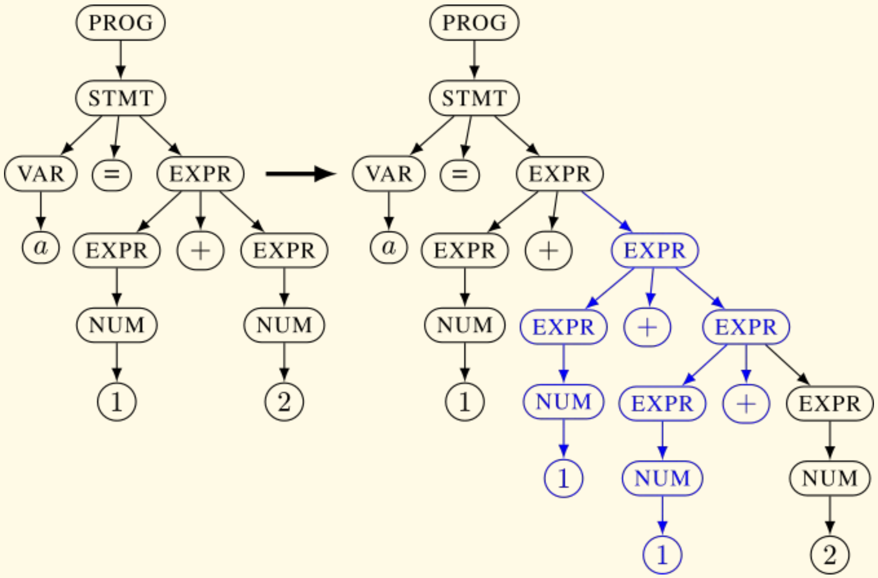
```
for i in range(10):
    print(simple_grammar_
```

```
7 / +48.5
-5.9 / 9 - 4 * +-(-+++((1
8.2 - 27 - -9 / +((+9 * --
* +-(((-(-6) * ---+6)) / +
0)))) * -+5 + 7.513)))) -
1 - -3 * 7 - 28 / 9
(+9) * +-5 * ++-926.2 - (+
8 + -(9.6 - 3 - -+-4 * +77
-(((((++((((+((++++-((+-37
* 0))) - 4 + +1
5.43
(9 / -405 / -23 - +-((+-(2
-++2 - (--+715769550) / 8
```

```
) / +(-+---((5.6 - --(3 * -1.8 * +(6
 (1 / +++6.37) + (1) / 482) / +++-+
 --++090

5)

* (1) / (-7.6 * 535338) + +256) * 0)
```



Nautilus Grammer Mutator

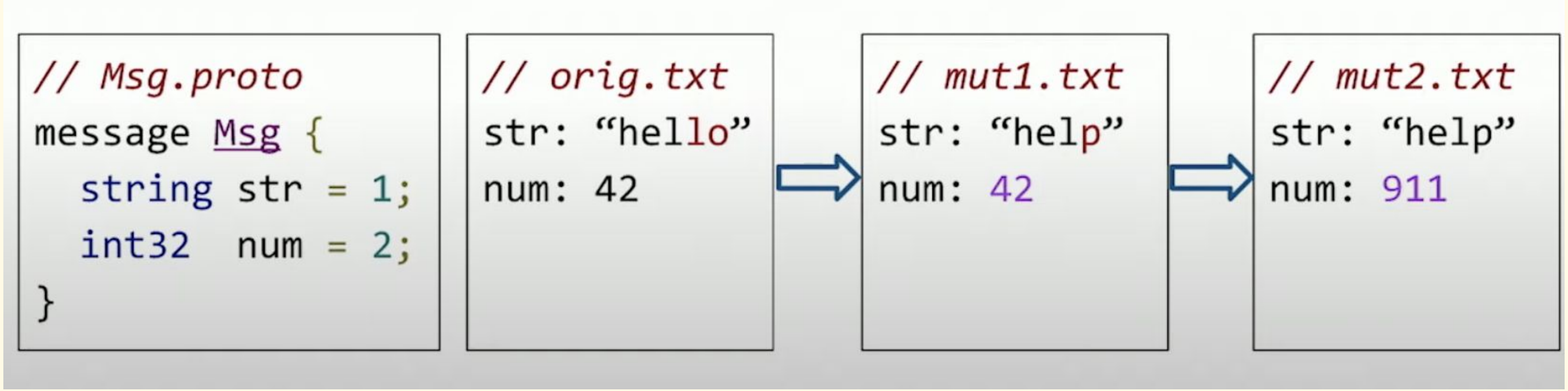



```
#ctx.rule(NONTERM: string, RHS: string|bytes) adds a rule
NONTERM->RHS. We can use {NONTERM} in the RHS to request a
recursion.
ctx.rule("START","<document>{XML_CONTENT}</document>")
ctx.rule("XML_CONTENT","{XML}{XML_CONTENT}")
ctx.rule("XML_CONTENT","")

#ctx.script(NONTERM:string, RHS: [string]], func) adds a rule
NONTERM->func(*RHS).
# In contrast to normal `rule`, RHS is an array of nonterminals.
# It's up to the function to combine the values returned for the
NONTERMINALS with any fixed content used.
ctx.script("XML",["TAG","ATTR","XML_CONTENT"], lambda
tag,attr,body: b"<%s %s>%s</%s>"%(tag,attr,body,tag) )
ctx.rule("ATTR","foo=bar")
ctx.rule("TAG","some_tag")
ctx.rule("TAG","other_tag")

#sometimes we don't want to explore the set of possible inputs in
more detail. For example, if we fuzz a script
#interpreter, we don't want to spend time on fuzzing all different
variable names. In such cases we can use Regex
#terminals. Regex terminals are only mutated during generation,
but not during normal mutation stages, saving a lot of time.
#The fuzzer still explores different values for the regex, but it
won't be able to learn interesting values incrementally.
#Use this when incremantal exploration would most likely waste
time.

ctx.regex("TAG","[a-z]+")
```

Protobuf Based - Libprotobuf

```cpp
class MyProtobufMutator : public
protobuf_mutator::Mutator {
 public:
  // Optionally redefine the Mutate* methods to
perform more sophisticated mutations.
}
void Mutate(MyMessage* message) {
  MyProtobufMutator mutator;
  mutator.Seed(my_random_seed);
  mutator.Mutate(message, 200);
}
```

```python
from person_pb2 import Person
p = Person()
p.name = "Alice"
p.id = 123
p.email = "alice@example.com"
------
data = p.SerializeToString()
p2 = Person()
p2.ParseFromString(data)
print(p2.name, p2.id, p2.email)
```



Snapshot Fuzzing

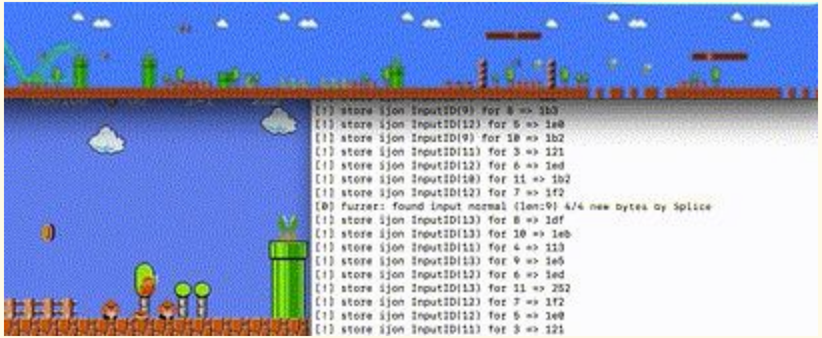
The idea here is that you are creating an image of the fuzzing target and then for every execution you take a snapshot of the the memory state, register state, or map data in the case of the example.

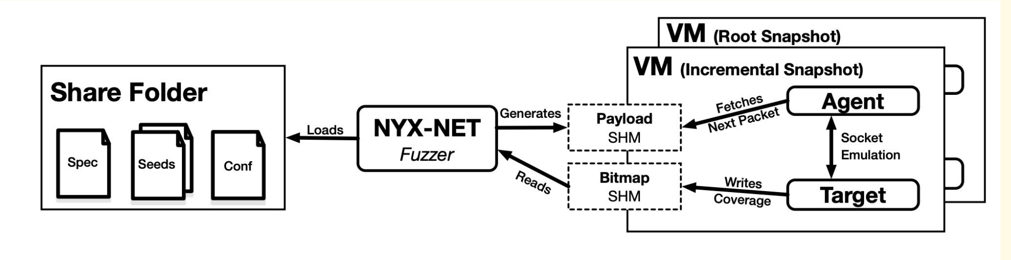
This allows you to fuzz on a specific state,

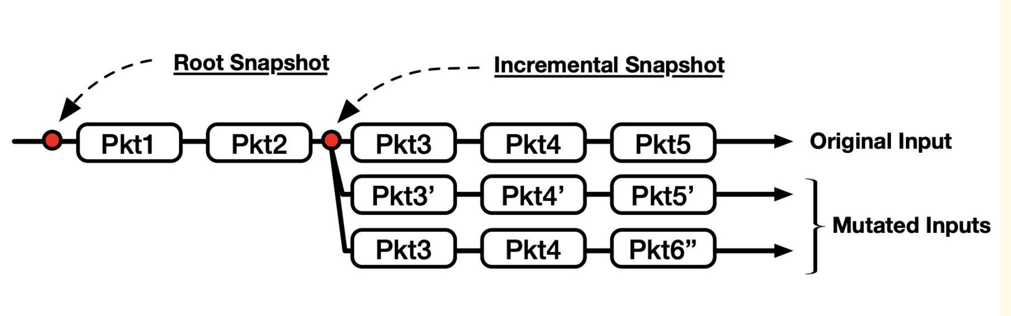
- For example a fork of NYX called nyx-net works by creating snapshots of network entry points of any network based service. A custom kernel is written in order to allow for direct fuzzing input access into network entry point like
    - Socket lifecycle: socket(), accept(), connect(), close(), dup()
    - Data transfer: read(), recv(), recvfrom(), send(), write()
    - Event polling: select(), poll(), epoll() and related variants

It's not a perfect system there are issues

- For fuzzing windows blackbox you have to snapshot the system of the actual VM not just the actual windows environment







Frida Fuzzing Black Box/Gray Box

The basis of fuzzing is the underlying idea that we can gain coverage information on the basics blocks and there edges. AS such it makes a lot of sense you use a dynamic analysis engine that work entirely black box to fuzz things we don't directly have access to asidece form a compiled binary.

At the start of the mode we can see that there are 2 main modules we are using throughout - Frida Gum and FridaRuntime. Frida Gum gives us access to two key APIs, the InstructionWriter which allows us to write ASM into the basic blocks of our target and the StalkerOutput a wrapper around the Stalker engine.

We start by providing our custom CoverageRuntime struct the FridaRuntime trait, adding the method init which takes the actual _gum: &frida_gum::Gum as parameters the primary API used for dynamic instrumentation, but the actual method itself is empty this holds true for the rest of the functions we see here and server to satisfy our traits inheritance.

```
impl FridaRuntime for CoverageRuntime {
    /// Initialize the coverage runtime
    /// The struct MUST NOT be moved after this function
is called, as the generated assembly references it
    fn init(
        &mut self,
        _gum: &frida_gum::Gum,
        _ranges: &RangeMap<u64, (u16, String)>,
        _module_map: &Rc<ModuleMap>,
    ) {
    }

    fn deinit(&mut self, _gum: &frida_gum::Gum) {}

    fn pre_exec(&mut self, _input_bytes: &[u8]) ->
Result<(), libafl::Error> {
        Ok(())
    }

    fn post_exec(&mut self, _input_bytes: &[u8]) ->
Result<(), libafl::Error> {
        Ok(())
    }
}
```

Looking Into Fuzzilli

Fuzzilli's central goal is generating "interesting" javascript code. However, because Fuzzilli targets core interpreter bugs, e.g in JIT compilers, it faces 2 core problems: *syntactical correctness* and *semantic correctness*.

Syntactical correctness:

Syntactically invalid JS will be rejected early on during processing in the engine by the parser (not a target). Fuzzili's solution is FuzzIL which can only express syntactically valid JS code.

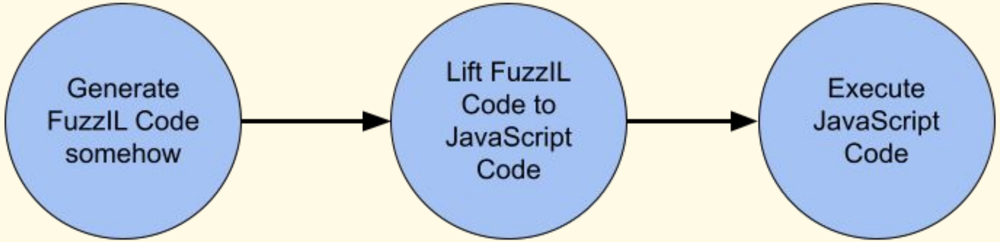
```
// syntactically incorrect code
// missing parentheses in multiple locations
function foo(n {
    let a = 0, b = 1;

    for let i = 0; i < n; i++ {
        console.log(a);
        let temp = a + b;
        a = b;
        b = temp;
    }
}
```

Semantic Correctness:

In Fuzzilli, a program that raises an uncaught exception is considered to be semantically incorrect, or simply invalid. This is in contrast to other scenarios, e.g. fuzzing runtime APIs, where semantic correctness can be worked around by wrapping generated code in try-catch constructs. While this is still possible for Fuzzilli, it will fundamentally change the control flow of the generated program and thus how it is optimized by a JIT compiler. This challenge is left up to each fuzzing engine.

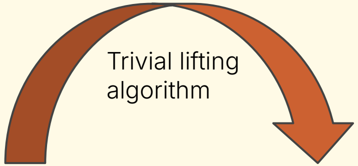
```
// semantically incorrect code
function calculateAverage(a, b, c) {
    // The intention is to calculate the
average of three numbers,but the logic is
wrong (missing division by 3)
    return a + b + c;
}
```

FuzzIL



Fuzzilli exclusively operates on FuzzIL programs internally and only lifts them to JS for execution.

A FuzzIL program is simply a list of instructions, e.g the imaginary FuzzIL sample below:




```
v0 <- BeginPlainFunctionDefinition -> v1,
v2, v3
    v4 <- BinaryOperation v1 '+' v2
    SetProperty v3, 'foo', v4
EndPlainFunctionDefinition
v5 <- LoadString "Hello World"
v6 <- CreateObject ['bar': v5]
v7 <- LoadFloat 13.37
v8 <- CallFunction v0, [v7, v7, v6]
```

```
function f0(a1, a2, a3) {
    const v4 = a1 + a2;
    a3.foo = v4;
}
const v5 = "Hello World";
const v6 = {bar: v5};
const v7 = 13.37;
const v8 = f0(v7, v7, v6);
```

FuzzIL

Since FuzzIL programs are immutable, when they are mutated, they're copied while mutations are applied. This is done in ProgramBuilder.

The Program class represents an immutable unit of code to which various operations can be applied.

```
public final class Program:
CustomStringConvertible {
        …

        /// The immutable code of this program.
        public let code: Code
         …

        /// Each program has a unique ID to identify
it even across different fuzzer instances.
        public private(set) lazy var id = UUID()

        public init() {
                self.code = Code()
                self.parent = nil
        }

        …

}
```

```
/// Builds programs.
///
/// This provides methods for constructing and appending random
/// instances of the different kinds of operations in a program.

public class ProgramBuilder {
        /// The fuzzer instance for which this builder is active.
        public let fuzzer: Fuzzer
        …

        /// The code and type information of the program that is
being constructed.
        private var code = Code()
        …

        /// Visible variables management.
        private var scopes = Stack<[Variable]>([[]])

        …

        /// Type inference for JavaScript variables.
        private var jsTyper: JSTyper

        …
}
```

FuzzIL Mutations

// Fundamental mutations..
Input Mutator ⇒ It replaces an input to an instruction with another, randomly chosen one.

Operation Mutator ⇒ Mutates the parameters of an operation, e.g. a '+' to a '/'.

Splicing ⇒ Copy a self-contained part of one program (or the whole program) into another in order to combine features.

Code Generation ⇒ Generates new, random code at one or multiple random positions in the mutated program.

// Runtime assisted mutations..
Exploration ⇒ Uses runtime type information available in JS to determine "useful" actions that can be performed on an existing value

Probing ⇒ Runtime-assisted mutator which tries to determine what properties exist within an object (rather it's prototype)

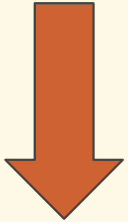
FixupMutator ⇒ Fix/improve existing code. Currently it only removes unnecessary try-catch blocks and guards

Base Instruction Mutator

```swift
/// Base class for mutators that operate on or at single instructions.
public class BaseInstructionMutator: Mutator {
      …

      /// Overridden by child classes.
      /// Determines the set of instructions that can be mutated by this mutator
      public func canMutate(_ instr: Instruction) -> Bool {
                  fatalError("This method must be overridden")
      }

      /// Overridden by child classes.
      /// Mutate a single statement
      public func mutate(_ instr: Instruction, _ builder: ProgramBuilder) {
                  fatalError("This method must be overridden")
      }

      …
}
```

Mutations: Input Mutator

```
SetProperty v3, 'foo', v4
```



```
SetProperty v3, 'foo', v2
```

```
public class InputMutator: BaseInstructionMutator {

      …

      public override func mutate(_ instr: Instruction, _ b:
ProgramBuilder) {

            let selectedInput = Int.random(in: 0..<instr.numInputs)
            let replacement: Variable?

            … get random replacement …

            if let replacement = replacement {
                  inouts[selectedInput] = replacement

                  b.append(Instruction(instr.op, inouts: inouts, flags:
            .empty))
        }
}
```

Mutations: Operation Mutator

```
v4 <- BinaryOperation v1 '+' v2
```

```
v4 <- BinaryOperation v1 '/' v2
```

```
public class OperationMutator: BaseInstructionMutator {
      …

      public override func mutate(_ instr: Instruction, _ b: 
ProgramBuilder) {
            let newInstr: Instruction

            if instr.isOperationMutable && instr.isVariadic {
                  … Code A …
            } else if instr.isOperationMutable {
                  newInstr = mutateOperation(instr, b)
            } else {
                  … code B …
        }

        b.adopt(newInstr)
    }

      private func mutateOperation(_ instr: Instruction, _ b: 
ProgramBuilder) -> Instruction {
            …
      }
}
```

Mutations: Splicing

```
v0 <- LoadInt '42'
v1 <- LoadFloat '13.37'
v2 <- LoadBuiltin 'Math'
v3 <- CallMethod v2, 'sin',
[v1]
v4 <- CreateArray [v3, v3]
```

```
... existing code
v13 <- LoadFloat '13.37'
v14 <- LoadBuiltin
'Math'
v15 <- CallMethod v14,
'sin', [v13]
... existing code
```

```
public class SpliceMutator: BaseInstructionMutator {
    private var deadCodeAnalyzer = DeadCodeAnalyzer()

    public override func beginMutation(of program: Program) {
        deadCodeAnalyzer = DeadCodeAnalyzer()
    }

    public override func mutate(_ instr: Instruction, _ b:
ProgramBuilder) {
        switch instr.op.opcode {
        case .wasmEndTypeGroup:
            b.buildIntoTypeGroup(endTypeGroupInstr: instr, by:
.splicing)
        default:
            b.adopt(instr)
            b.build(n: defaultCodeGenerationAmount, by:
.splicing)
        }
    }
}
```

Mutations: Splicing

```
private func buildInternal(initialBuildingBudget: Int, mode:
BuildingMode) {
      …
      switch mode {
      …
      case .splicing:
            let program = fuzzer.corpus.randomElementForSplicing()
         …
            splice(from: program)
         …
      }
      …
}

@discardableResult
public func splice(from program: Program, at specifiedIndex: Int? =
nil, mergeDataFlow: Bool = true) -> Bool {
      …
}
      …
}
```

Mutations: Code Generation

```
/// Possible building modes. These are
used as argument for build() and
determine how the new code is
produced.
    public enum BuildingMode {
        // Generate code by running
CodeGenerators.
        case generating
        // Splice code from other
random programs in the corpus.
        case splicing
        // Do all of the above.
        case generatingAndSplicing
    }
```

```
public class CodeGenMutator: BaseInstructionMutator {
    …

    public override func mutate(_ instr: Instruction, _
b: ProgramBuilder) {
        switch instr.op.opcode {
        case .wasmEndTypeGroup:
            …
        default:
            b.adopt(instr)
            b.build(n: defaultCodeGenerationAmount,
by: .generating)
        }
    }

}
```

Mutations: Code Generation

```
public class ProgramBuilder {

public func build(n: Int = 1, by mode: BuildingMode = .generatingAndSplicing) {
      …
      buildInternal(initialBuildingBudget: n, mode: mode)
      …
}

private func buildInternal(initialBuildingBudget: Int, mode: BuildingMode) {
      …
      if state.mode != .splicing {
            availableGenerators = fuzzer.codeGenerators.filter({
$0.requiredContext.isSubset(of: origContext) })
            …
        }

      switch mode {
      case .generating:
            …
            let generator = availableGenerators.randomElement()
            …
            run(generator)
      }
      …
}
```

Runtime Assisted Mutations

```
/// A mutator that uses runtime feedback to perform smart(er) mutations.
///
/// A runtime assisted-mutator will generally perform the following steps:
/// 1. Instrument the program to mutate in some way, usually by inserting special operations.
/// 2. Execute the instrumented program and collect its output through the fuzzout channel.
/// 3. Process the output from step 2. to perform smarter mutations and generate the final program.
///
/// See the ExplorationMutator or ProbingMutator for examples of runtime-assisted mutators.
public class RuntimeAssistedMutator: Mutator {
        …

        // Instrument the given program.
        func instrument(_ program: Program, for fuzzer: Fuzzer) -> Program? {
                fatalError("Must be overwritten by child classes")
        }

        // Process the runtime output of the instrumented program and build the final program from that.
        func process(_ output: String, ofInstrumentedProgram instrumentedProgram: Program, using b: ProgramBuilder) ->
(Program?, Outcome) {
                fatalError("Must be overwritten by child classes")
        }

        …
}
```

Mutations: Exploration

High Level Steps:

1. Instrument the given program by inserting “Explore” operations for existing variables

2. Lift and execute these Explore operations to a chunk of code that inspects the variables at runtime and select “useful” actions that are reported back to fuzzilli.

3. The mutator processes the output of step 2 and replaces the Explore operations with the concrete action that was performed by them at runtime.

Mutations: Exploration

```
public class ExplorationMutator: RuntimeAssistedMutator {
        …
        override func instrument(_ program: Program, for fuzzer: Fuzzer) -> Program? {
                // Enumerate all variables in the program and put them into one of two buckets, depending on whether static type information is available for them.
                var untypedVariables = [Variable]()
                var typedVariables = [Variable]()
                for instr in program.code {
                        …
                }
                var pendingExploreStack = Stack<Variable?>()
                b.adopting(from: program) {
                        for instr in program.code {
                                …
                                for v in instr.outputs where variablesToExplore.contains(v) {
                                        if instr.isBlockStart {
                                                …
                                                pendingExploreStack.top = v
                                        } else {
                                                explore(v)
                                        }
                                }
                                for v in instr.innerOutputs where variablesToExplore.contains(v) {
                                        explore(v)
                                }
                        }
                }
                …
                return instrumentedProgram
        }
        …
}
```

Mutations: Exploration

Note: A large amount of this mutator code in located in the lifter code that implements Explore operations in the target language. For JS, the logic is located in JavaScriptExploreLifting.swift

```
override func process(_ output: String, ofInstrumentedProgram instrumentedProgram: Program, using b: ProgramBuilder) -> (Program?, Outcome) {
        for line in output.split(whereSeparator: \.isNewline) where line.starts(with: "EXPLORE") {
                … look for/handle errors …
        }
        // Now build the real program by replacing every Explore operation with the operation(s) that it actually performed at runtime.
        b.adopting(from: instrumentedProgram) {
                for instr in instrumentedProgram.code {
                        if let op = instr.op as? Explore {
                                if let entry = actions[op.id], let action = entry {
                                        …
                                        let exploredValue = b.adopt(instr.input(0))
                                        let args = instr.inputs.suffix(from: 1).map(b.adopt)
                                        …
                                        do {
                                                let context = (arguments: args, specialValues: ["exploredValue": exploredValue])
                                                try action.translateToFuzzIL(withContext: context, using: b)
                                        } catch ActionError.actionTranslationError(let msg) {
                                                logger.error("Failed to process action: \(msg)")
                                        } catch {
                                                logger.error("Unexpected error during action processing \(error)")
                                        }
                                        …
                                }
                        } else {
                                d.adopt(instr)
                        }
                }
        }
        return (b.finalize(), .success)
}
```

Mutations: Probing

Note: A large amount of this mutator code in located in the lifter code that implements Explore operations in the target language. For JS, the logic is located in JavaScriptProbeLifting.swift

High Level Steps:

1. Instrument the given program by inserting Prob operations which turns an existing variable into a "probe". The probe records accesses to *non-existent* properties on the original value. Really, the object's prototype is replaced with a Proxy.

2. Lift and execute the instrumented program which then reports the property names of non-existent accesses back to fuzzilli.

3. The mutator processes the output of step 2 and randomly selects properties to install. It then converts the Probe operations into an appropriate FuzzIL operation.

Mutations: Probing

```
public class ProbingMutator: RuntimeAssistedMutator {
        …
        override func instrument(_ program: Program, for fuzzer: Fuzzer) -> Program? {
                // Determine candidates for probing: every variable that is used at least once as an input is a candidate.
                var usedVariables = VariableSet()
                for instr in program.code {
                        …
                        usedVariables.formUnion(instr.inputs)
                }
                let candidates = Array(usedVariables)

                var pendingProbesStack = Stack<Variable?>()
                let b = fuzzer.makeBuilder()
                b.adopting(from: program) {
                        for instr in program.code {
                                b.adopt(instr)

                                … only probe block ends  …

                                for v in instr.innerOutputs where variablesToProbe.contains(v) {
                                        b.probe(v, id: v.identifier)
                                }

                                … only probe block ends …
                        }
                }
                …
                return instrumentedProgram
        }
}
```

Mutations: Probing

```
override func process(_ output: String, ofInstrumentedProgram instrumentedProgram: Program, using b: ProgramBuilder) -> (Program?,
RuntimeAssistedMutator.Outcome) {
        …
        var results = [String: Result]()
        for line in output.split(whereSeparator: \.isNewline) where line.starts(with: "PROBING") {

                … look for errors …

                let decoder = JSONDecoder()
                        let payload = Data(line.dropFirst(resultsMarker.count).utf8)
                guard let decodedResults = try? decoder.decode([String: Result].self, from: payload) else {
                                        … handle errors …
                }
                         results = decodedResults
        }
        // Now build the final program by parsing the results and replacing the Probe operations
        // with FuzzIL operations that install one of the non-existent properties (if any).
        b.adopting(from: instrumentedProgram) {
                for instr in instrumentedProgram.code {
                        if let op = instr.op as? Probe {
                                if let results = results[op.id] {
                                        let probedValue = b.adopt(instr.input(0))
                                        …
                                        processProbeResults(results, on: probedValue, using: b)
                                        …
                                }
                         } else {
                            b.adopt(instr)
                         }
                }
        }
        return (b.finalize(), .success)
}
```

Mutations: FixupMutator

High Level Steps:

1. Convert “fixable instructions” into JS Actions
2. Executed the instrument program
3. On the JS side, when executing a Fixup operation, inspect the associated action and determine if it can/should be modified.
4. Send the modified Actions back to Fuzzilli which then replaces the Fixup instructions with the potentially modified Actions

Mutations: FixupMutator

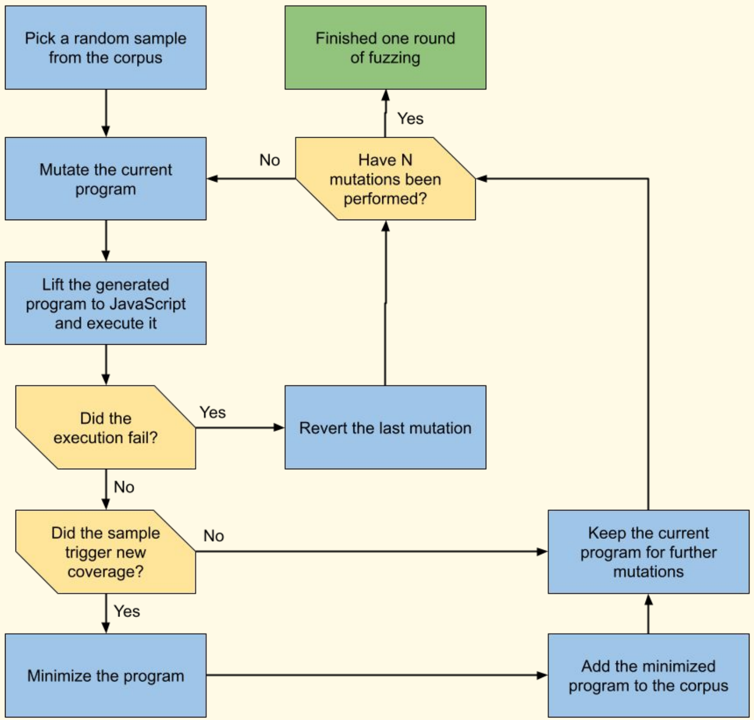
```
public class FixupMutator: RuntimeAssistedMutator {
        override func instrument(_ program: Program, for fuzzer: Fuzzer) -> Program? {
        …
                func fixup(_ instr: Instruction, performing op: ActionOperation, guarded: Bool, withInputs inputs: [Action.Input], with b: ProgramBuilder) {
                        …
                        numInstrumentedInstructions += 1

                                let id = "instr\(instr.index)"
                                let action = Action(id: id, operation: op, inputs: inputs, isGuarded: guarded)
                                let encodedData = try! actionEncoder.encode(action)
                                let encodedAction = String(data: encodedData, encoding: .utf8)!
                                let maybeOutput = b.fixup(id: id, action: encodedAction, originalOperation: instr.op.name, arguments: Array(instr.inputs),
hasOutput: instr.hasOneOutput)
                        …
                }
                …
                func fixupIfGuarded(_ instr: Instruction, performing op: ActionOperation, guarded: Bool, withInputs inputs: [Action.Input], with b:
ProgramBuilder) {
                                …
                                fixup(instr, performing: op, guarded: guarded, withInputs: inputs, with: b)
                }

                for instr in program.code {
                        switch instr.op.opcode {
                                case .callFunction(let op):
                                                let inputs = (0..<instr.numInputs).map({ Action.Input.argument(index: $0) })
                                                fixupIfGuarded(instr, performing: .CallFunction, guarded: op.isGuarded, withInputs: inputs, with: b)

                                … handle more cases …
                        }
                }
                …
                return instrumentedProgram
        }
}
```

Mutations: FixupMutator

```
override func process(_ output: String, ofInstrumentedProgram instrumentedProgram: Program, using b: ProgramBuilder) -> (Program?,
RuntimeAssistedMutator.Outcome) {
        …

        // Now build the real program by replacing every Fixup operation with either the new (if we got one) or original Action.
                for instr in instrumentedProgram.code {
                if let op = instr.op as? Fixup {
                        let args = Array(instr.inputs)
                        do {
                                        try action.translateToFuzzIL(withContext: (arguments: args, specialValues: [:]), using: b)

                                } catch ActionError.actionTranslationError(let msg) {
                                … handle error …
                                        } catch {
                                                …
                        }

                } else {
                                        b.append(instr)
                        }

        }

        …
}
```

Mutation Engine

```
for _ in 0..<maxAttempts {
      if let result = mutator.mutate(parent, for:
fuzzer) {
            // Success!
           result.contributors.formUnion(parent.
contributors)

            mutator.addedInstructions(result.size -
      parent.size)
           mutatedProgram = result
           Break
      } else {
             // Try a different mutator.
            mutator.failedToGenerate()
            mutator =  fuzzer.mutators.randomElement()
      }
}
```



Mutation Engine performs N mutations on the selected program and looks for new coverage or semantic errors. A major limitation of MuationEngine is that it will struggle to find bugs/programs that require a specific sequence of N mutations from the original program.

Type System and Type Inference

Fuzzilli's **type system** and **type inference** are optimizations that improve the effectiveness of its code generators and mutators by reducing the generation of invalid code. By inferring the types of variables, the fuzzer can produce more realistic and valid programs, leading to more successful fuzzing runs.

```
CodeGenerator("FunctionCallGenerator")
{ b in
    let f = b.randomVariable()
    let arguments =
[b.randomJsVariable(),
b.randomJsVariable(),
b.randomJsVariable()]
    b.callFunction(f, with: arguments)
}
```

Without Type Inference

```
v3 <- LoadString "foobar"
v4 <- CallFunction v3, []
// TypeError: v3 is not a
function
```

Fuzzilli's Core Components

- MutationFuzzer: produces new programs from existing ones by applying mutations. Afterwards executes the produced samples and evaluates them.
- ScriptRunner: executes programs of the target language.
- Corpus: stores interesting samples and supplies them to the core fuzzer.
- Environment: has knowledge of the runtime environment, e.g. the available builtins, property names, and methods.
- Minimizer: minimizes crashing and interesting programs.
- Evaluator: evaluates whether a sample is interesting according to some metric, e.g. code coverage.
- Lifter: translates a FuzzIL program to the target language (JavaScript).

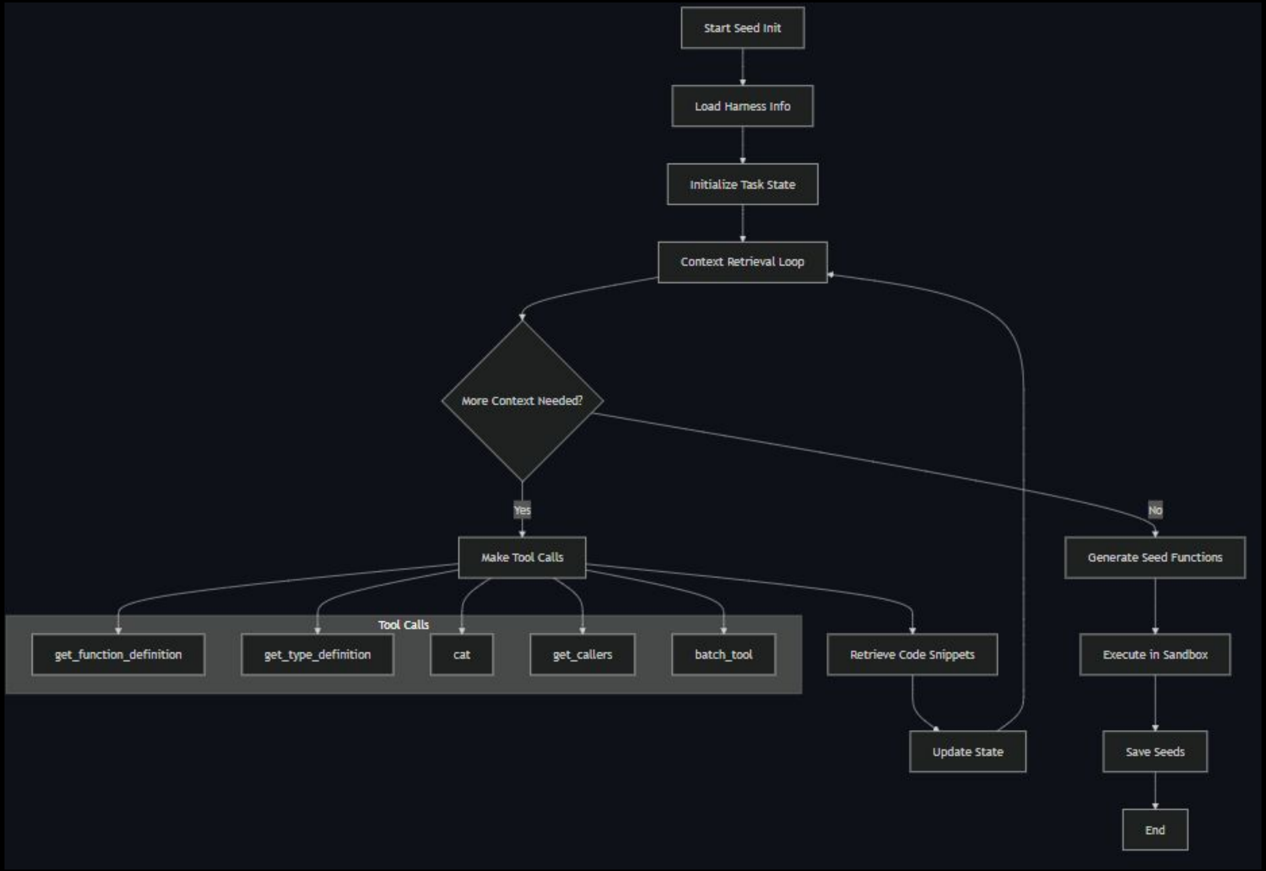
ToB AIxCC Seed Generator

- Seed initialization
  - Generates the initial seed inputs for bootstrapping a fuzzing corpus. These are ‘example’ input that a harness uses to start exploration.
  - Goal: get good coverage early so subsequent fuzzing is more effective.

- Seed exploration
  - Once you have the seeds from initialization, this module will use fuzzing/mutation, coverage feedback, crashes, etc.. to discover new behaviors of the program.

- Vulnerability discovery
  - After exploration reveals paths, behaviors, and input shapes, this module will use crashes found by fuzzers, sanitizers, symbolic exec, static analysis, etc.. to detect issues caused by certain inputs.
  - The bugs it finds are then used to generate PoVs which reproduce the crash and demonstrate the vuln is reachable and exploitable

Seed Initialization

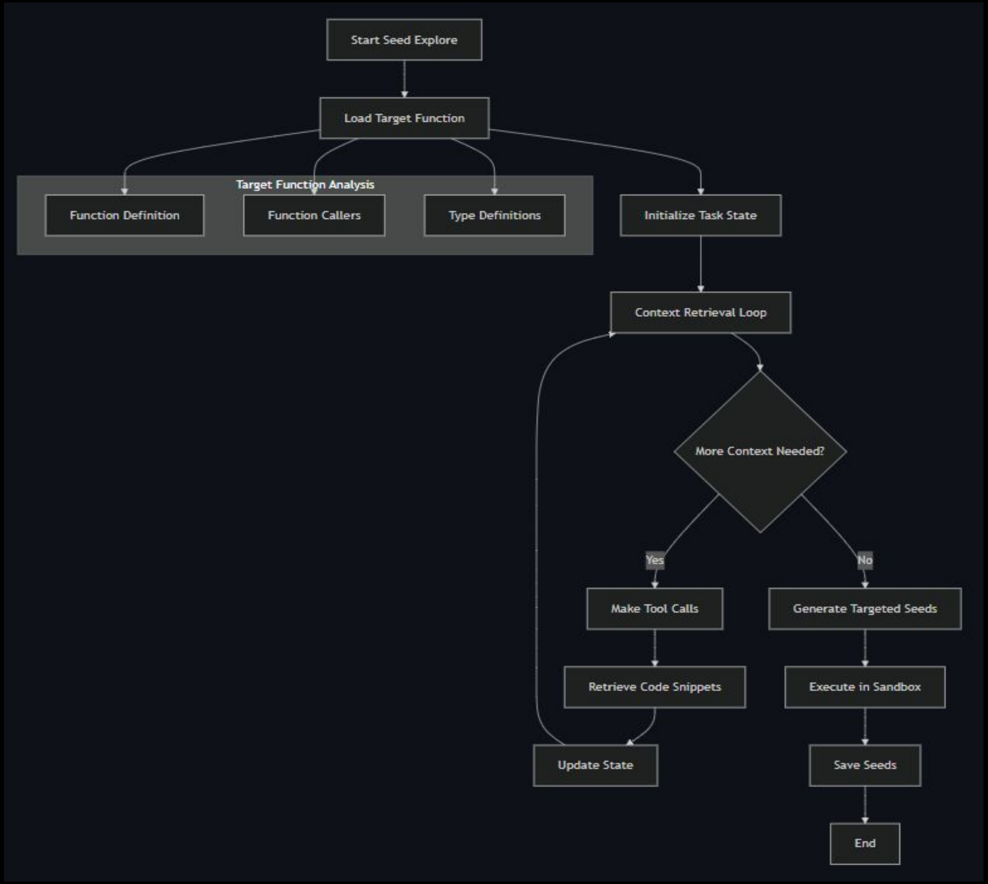


- Captures the harness and task information from task system

  Gains context by making various tool calls to a CodeQuery database such as get_func_def, get_type_def etc.

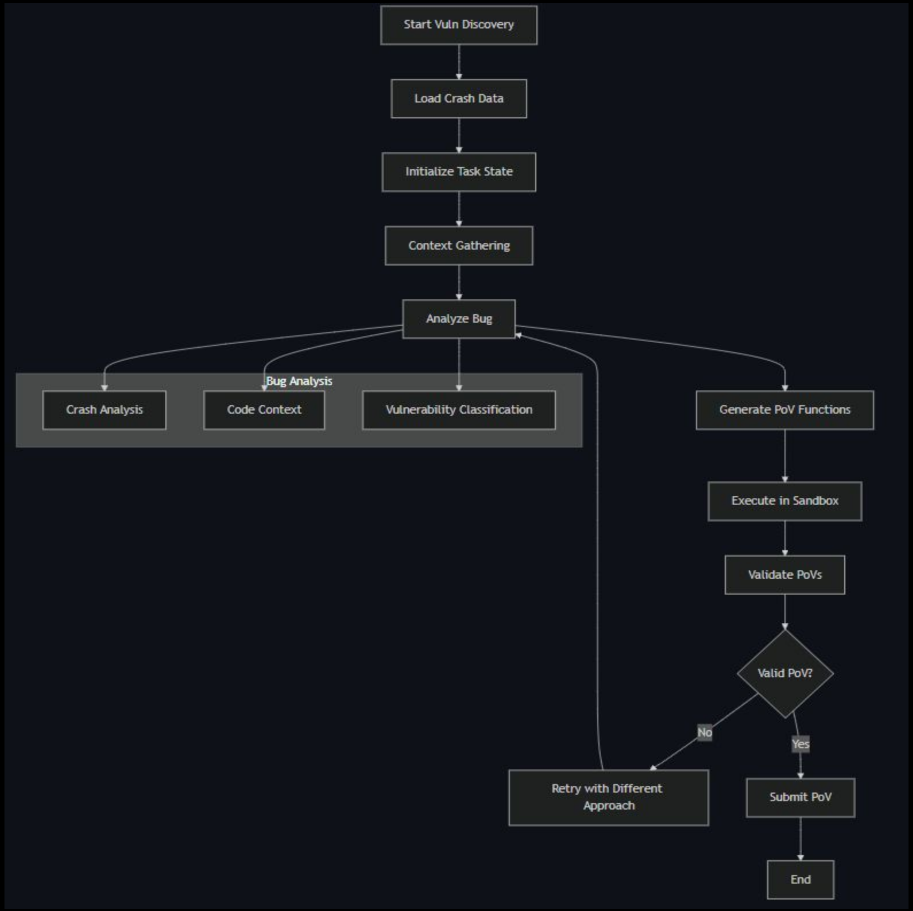
- When it has sufficient context it passes that into various seed generation functions and uses the result to begin a campaign*

Seed exploration



- Uses the seeds from the initializer as inputs to harnesses and assesses the returned feedback.
- Based on the feedback it makes targeted mutations that can reach new code paths or other forms of output like crashes.
- Note: these are specific to each harness of course.

Vulnerability Discover



- Called whenever the exploration step results in a crash/unusual output.
- Loads the crash data and execution state
- Uses that information to analyze the bug and generate Proof of Vulnerability. If the PoV isn't sufficient it starts the process again

The seedgen component is responsible for automatically generating seed inputs. It coordinates closely with the task system, which manages the overall workflow: producing seeds, launching fuzzing or exploration campaigns, invoking static-analysis and modeling tools, and triggering vulnerability discovery once anomalous behavior is found.

The task system decides when seedgen should perform seed initialization, when to run exploration (e.g. fuzzers), when to run static analyzers or code modeling tools, and when to launch vulnerability discovery tasks.

Part of this orchestration involves fuzzy matching, which helps to reduce duplication among inputs, and to map inputs to their coverage / behavior so that seed selection is more efficient. Fuzzy matching will match based on strings instead of binary patterns. For example, it will map 'trail' to 'trial'. Further matching would be something like 'Martin Luther Junior' to 'Martin Luther King, Jr.'.