Language-Based Engineering:

A Comprehensive Approach to Software Analysis and Hardening

Hyunsoo Shin

Program Coverage

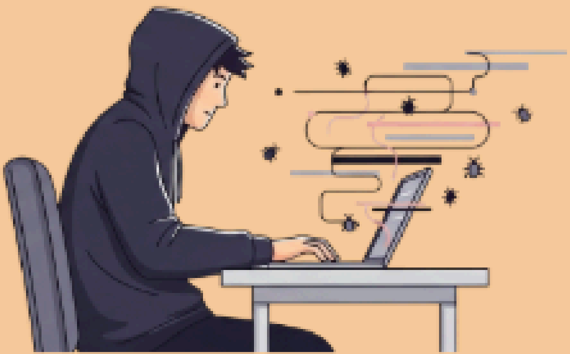
Buffer Overrun:
Address Sanitizer

Fuzzer

Delta Debugging

Symbolic
Execution

Data Race
Detector



LLM-based Fuzzer

First Print
08-25-2025

PUBLISHED BY HYUNSOO SHIN

(“ https://github.com/flavio20002/typst-orange-template “, “TEMPLATE-WEBSITE”)

First printing, August 2025

Buffer Overrun: Address Sanitizer

Program Coverage

Fuzzer

Symbolic Execution

LLM-based Fuzzer

Data Race Detector

Delta Debugging

Contents

Buffer Overrun: Address Sanitizer
Program Coverage
LLM-based Fuzzer
Fuzzer
Symbolic Execution
Data Race Detector
Delta Debugging

List of Figures

Buffer Overrun: Address Sanitizer
Symbolic Execution
Program Coverage
Fuzzer
LLM-based Fuzzer
Data Race Detector
Delta Debugging

List of Tables

Part I

Introduction to Language-based engineering

```
printf("Hello, World");
```

1. Welcome aboard

1.1 As a Programmer

As usual, I was coding and Googling to find the traditional use cases of a topic I had just encountered for the first time. I clicked on one of the top search results—a blog—and on its front page, I saw a phrase written in Korean: "개발자로 살길 정말 잘했다" ("I'm really glad I chose to be a programmer").

Yes, I had forgotten the joy of programming for some time. I realized I'd been too focused on building programs quickly and forcing myself to study academic topics. I set high standards for myself in an effort to become a great programmer, but it ended up making me accumulate knowledge under stress and pressure. No one was pushing me to do that — I'm not even sure why I was. Maybe it's just in my DNA.

Writing this book turned out to be a stroke of luck. It brought back the fragrance of joy that programming once gave me. Honestly, I have to admit — while writing this book, there were times I was just focused on finishing it as quickly as possible. Looking back, I must have been a little crazy.

I truly hope this book brings you the same sense of relaxation and enjoyment I felt while writing it. Enjoy!

1.2 About the book

There are a lot of research area in computer science field such as Artifical Intelligence, Network, HCI (Human-Computer-Interaction), Security, PL (Programming Language), Computer Architecture, Operating System, Computer Graphics, Computer Vision, Robotics, Database, Distributed System, Cryptography, etc.

Personally, I don't prefer research areas that require additional equipment. On the contrary, I'm drawn to fields where I can do meaningful work with just a laptop. That's why programming languages (PL) really suit my personality — I never feel the need for extra resources to study it. However, I sometimes struggle with PL because it's deeply rooted in algorithms, mathematics, and logic. I've seen many brilliant minds in this field, and I often find motivation in their work.

Secondly, I've been interested in the field of security. Security itself is a broad area, but my focus has been on language-based security [1]; — that is, studying security through the lens of programming languages (PL). I'm also particularly interested in topics like optimization, abstraction, and correctness.

This book explores a series of common program bugs that many of us have likely encountered while coding. At its core, the goal is to identify bugs — from a security perspective — **using PL technologies**.

1.2.1 Contents

This book delves into seven major topics related to program analysis and software testing:

Topics

1. Coverage
   a. Reports line, branch, and function-level coverage during program execution.
2. Buffer Overrun: Address Sanitizer
   a. Detects out-of-bounds memory accesses on the heap and stack.
3. Fuzzing
   a. Develops a coverage-guided fuzzer inspired by AFL (American Fuzzy Lop).
4. Delta Debugging
   a. Minimizes crashing inputs generated during fuzzing by isolating the minimal failure-inducing input.
5. Symbolic Execution
   a. Solves branch conditions to enhance code coverage during fuzzing.
6. LLM-based Synthesis
   a. Explore LLM-based fuzzer and implement LLM-based synthesis to generate test code snippets
7. Data Race Detection
   a. Identifies data races in multi-threaded programs.

1.2.2 Source and Target Language

The implementations in this book are written in Rust. Theoretically, the target language is the C language family. During the writing of this book, only C was thoroughly tested, with limited experimentation in C++. However, the approach can be extended to any language that supports an LLVM IR backend.

Broad language support is achieved through the frontend, which is responsible for recognizing the intermediate representation (IR) and providing the corresponding handlers within the instrumentation module.

1.2.3 LLVM Version

The book uses LLVM version 17.0.6, which was released on November 28, 2023.

1.2.4 Prerequisite

Readers are expected to be familiar with Rust programming. This book does not cover Rust syntax, coding conventions, or other language fundamentals.

1.2.5 Environment

We provide a Nix environment that sets up LLVM 17 in your shell. Simply type `nix-shell` in the top-level directory where `default.nix` is located. This environment also includes all the dependencies required for the book.

1.2.6 Compile and test

The build recipe is defined in the justfile located in the top-level directory.

- `just build`: Compile the entire project
- `just test`: Run unit test

1.2.7 Rationale

The code examples provided in this book may contain bugs, typographical errors, or inaccurately explained concepts. If you identify any issues, contributions via pull requests are welcome and appreciated.

Three Principles

2. Prerequisit: Instrumentation, Runtime, and LLVM IR

2.1 Compiler Pass Exploting

Human writes a program using high-level syntax like:

```c
int sum = a + b;
```

Compiler converts this high-level expression into lower expression like:

```llvm
%a = load i32, ptr %1
%b = load i32, ptr %2
@sum = add i32 @a, @b
```

Last lowering is transformation into bytecode, which is machine independent such as x86 and AMD.

```
100010101010010...
```

From a programmer's perspective, the compilation process—from intermediate representation (IR) to bytecode—is typically abstracted away. Developers focus on writing high-level source code, which serves as input to the compiler. For instance, when attempting to measure code coverage, manually inserting function calls to track line execution is both tedious and error-prone. Moreover, this approach does not scale well, as it would require repeating the same effort for each program individually.

This is where instrumentation comes into play, introduced here for the first time in this book. An instrumentation tool automatically inserts function calls (e.g., for line hits) into the source or IR, relieving the programmer of manual effort. This process is typically performed within the compiler's middle layer. For example, Clang [2];supports provides an instrumentation framework based on LLVM Passes [3].

In this book, we implement instrumentation entirely in Rust as a modular system. The module takes IR as input, performs transformations, and outputs instrumented IR. Each instrumentation module can target a different concern, and multiple instrumentation layers can be applied sequentially—for example, instrumenting first with Module A, then B, and finally C.

A sample of instrumented IR is shown below:

```LLVM-IR
%a = load i32, ptr %1
%b = load i32, ptr %2
@sum = add i32 @a, @b
call line_hit(...) // instrumented
```

2.2 Runtime Library

Compilers generate executable code, which can be run directly on a CPU. However, there are inherent limitations to what can be achieved solely through statically generated code.

For example, memory allocation in C is performed using `malloc()`, a POSIX-standard interface [4]. The actual memory management is handled by the runtime library (e.g., glibc), which internally invokes system calls to the operating system. In modern programming languages, garbage collection automates memory management, relieving the programmer from manual allocation and deallocation. Like `malloc()`, garbage collectors are part of the runtime system, not the compiler's output.

Go (Golang) is a prominent example of a language that supports reflection [5]. At runtime, Go programs can inspect type information, examine generic values such as interface{} or any, and iterate over struct fields. These capabilities are enabled by the runtime library, not the compiler alone.

In summary, runtime libraries play a crucial role in extending the dynamic behavior of programming languages beyond what the compiler can provide.

In this book, we implement a custom runtime library as a shared library written in Rust. It interacts with the instrumented code during execution, enabling dynamic analysis and runtime features.

Having now introduced the two core components of this book—**instrumentation** and **runtime**—we're ready to dive into implementation.

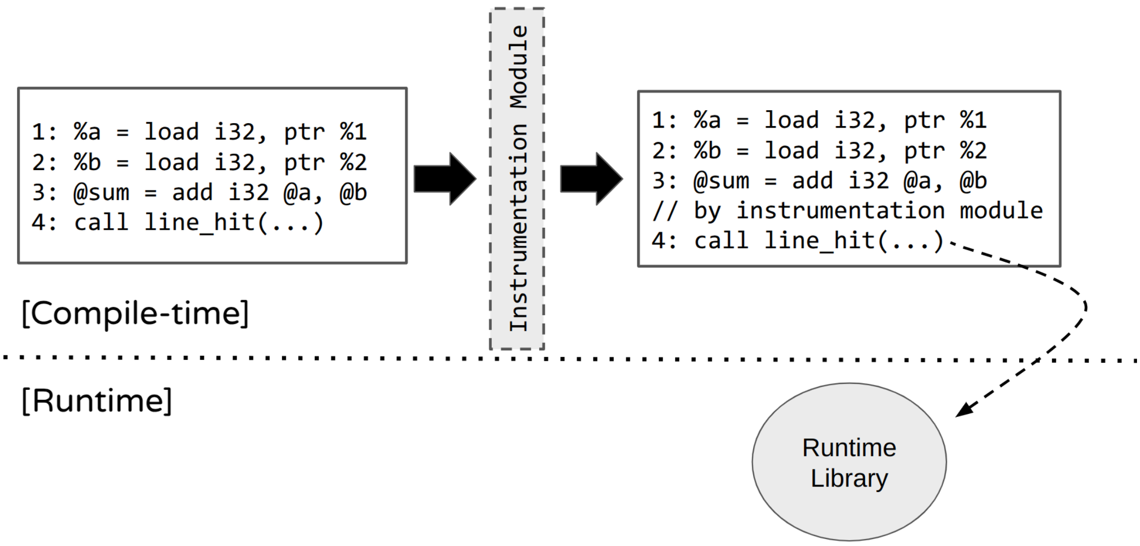


Figure 2.1: Concept of Module and Runtime

2.3 Benefit of IR

This chapter introduces the principles of IR, with a focus on LLVM IR. Before diving into IR itself, let's first ask: *Why is IR well-suited for program instrumentation?* Beyond instrumentation, IR is also a common target for tasks such as program analysis, correctness checking, and formal verification.

Let's take a look example C code.

```c
#include <stdio.h>

#define ARR_SIZE 5

int get_odd_even_diff(int* arr, int size) {
    int even_cnt = 0;
    int odd_cnt = 0;
    for (int i=0; i<size; i++) {
        if (arr[i] % 2 == 0) {
            even_cnt++;
        } else {
            odd_cnt++;
        }
    }
    return odd_cnt - even_cnt;
}

int main() {
    int arr[ARR_SIZE] = {1,2,3,4,5};
    printf("%d\n", get_odd_even_diff(arr, ARR_SIZE));
}
```

The function `get_odd_even_diff()` takes a pointer to an integer array along with its size. It counts the number of odd and even elements in the array, then returns the difference between the count of odd and even numbers. For example, given the array `{1, 2, 3, 4, 5}`, the function returns 1 because there are 3 odd numbers and 2 even numbers, and `3 - 2 = 1`.

Below is the corresponding LLVM IR code for the `get_odd_even_diff()` function.

```llvm
define i32 @get_odd_even_diff(ptr noundef %0, i32 noundef %1) #0 !dbg !16 {
  %3 = alloca ptr, align 8
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  %6 = alloca i32, align 4
  %7 = alloca i32, align 4
  store ptr %0, ptr %3, align 8
```

```
  call void @llvm.dbg.declare(metadata ptr %3, metadata !22, metadata !DIExpression()), !dbg !23
  store i32 %1, ptr %4, align 4
  call void @llvm.dbg.declare(metadata ptr %4, metadata !24, metadata !DIExpression()), !dbg !25
  call void @llvm.dbg.declare(metadata ptr %5, metadata !26, metadata !DIExpression()), !dbg !27
  store i32 0, ptr %5, align 4, !dbg !27
  call void @llvm.dbg.declare(metadata ptr %6, metadata !28, metadata !DIExpression()), !dbg !29
  store i32 0, ptr %6, align 4, !dbg !29
  call void @llvm.dbg.declare(metadata ptr %7, metadata !30, metadata !DIExpression()), !dbg !32
  store i32 0, ptr %7, align 4, !dbg !32
  br label %8, !dbg !33

8:                                                ; preds = %27, %2
  %9 = load i32, ptr %7, align 4, !dbg !34
  %10 = load i32, ptr %4, align 4, !dbg !36
  %11 = icmp slt i32 %9, %10, !dbg !37
  br i1 %11, label %12, label %30, !dbg !38

12:                                               ; preds = %8
  %13 = load ptr, ptr %3, align 8, !dbg !39
  %14 = load i32, ptr %7, align 4, !dbg !42
  %15 = sext i32 %14 to i64, !dbg !39
  %16 = getelementptr i32, ptr %13, i64 %15, !dbg !39
  %17 = load i32, ptr %16, align 4, !dbg !39
  %18 = srem i32 %17, 2, !dbg !43
  %19 = icmp eq i32 %18, 0, !dbg !44
  br i1 %19, label %20, label %23, !dbg !45

20:                                               ; preds = %12
  %21 = load i32, ptr %5, align 4, !dbg !46
  %22 = add i32 %21, 1, !dbg !46
  store i32 %22, ptr %5, align 4, !dbg !46
  br label %26, !dbg !48

23:                                               ; preds = %12
  %24 = load i32, ptr %6, align 4, !dbg !49
  %25 = add i32 %24, 1, !dbg !49
  store i32 %25, ptr %6, align 4, !dbg !49
```

```
  br label %26

26:                                               ; preds = %23, %20
  br label %27, !dbg !51

27:                                               ; preds = %26
  %28 = load i32, ptr %7, align 4, !dbg !52
  %29 = add i32 %28, 1, !dbg !52
  store i32 %29, ptr %7, align 4, !dbg !52
  br label %8, !dbg !53, !llvm.loop !54

30:                                               ; preds = %8
  %31 = load i32, ptr %6, align 4, !dbg !57
  %32 = load i32, ptr %5, align 4, !dbg !58
  %33 = sub i32 %31, %32, !dbg !59
  ret i32 %33, !dbg !60
}
```

The most striking aspect of IR is its fine-grained nature compared to high-level source code. For example, the condition `if (arr[i] % 2 == 0)` appears as a single line in high-level code, but in IR, it's broken down into multiple instructions:

- Access of array index operation
- A modulo operation
- An equality comparison
- A conditional branch

When working with high-level code, we would need to manually decompose and parse each operator for such fine-grained analysis.

Another key feature of IR is that variables are never reassigned. In LLVM IR, an instruction like `%1 = add i32 %a, %b` introduces a new variable `%1`. If another computation follows, it generates a new variable, such as `%2`, rather than reusing `%1`. This property is known as **Static Single Assignment (SSA)** form [6]. SSA greatly simplifies program analysis by making it easier to track the origin and use of each value. Let's look at an example.

```c
1 int a = 1; // scope 1
2 if (...) { // scope 2
3     int a = 2;
4     printf("%d\n", a); // the variable in line 3 of scope 2 is referenced here.
5 }
```

Without SSA, tracking variable scopes and updates becomes our responsibility. However, with SSA, this burden is lifted—we can rely on the SSA form to handle variable scoping automatically. As a result, program analysis becomes significantly simpler and more reliable.

```c
int a_1 = 1;
if (...) {
    int a_2 = 2;
    printf("%d\n", a_1);
}
```

Lastly, IR explicitly represents control flow. In the `get_odd_even_diff()` function, for example, if the for loop condition evaluates to false, execution jumps to line 11. Similarly, if the if condition on line 5 is false, control transfers to line 8. LLVM IR uses the concept of basic blocks, which are sequences of instructions with no internal jumps or branches—control only enters at the beginning and exits at the end. All high-level control structures (`if`, `for`, `while`, etc.) are translated into a series of basic blocks connected by branch instructions. Thanks to this structure, we don't need to manually determine the next program counter (PC); the IR makes control flow explicit and analyzable.

In summary, three key benefits highlight the importance of focusing our business logic on instrumentation, analysis, and related tasks.

- [x] Decomposed instructions
- [x] Control-flow
- [x] SSA

2.4 LLVM IR Tutorial

First, to generate IR file, we should add additional compiler flags `-S` and `-emit-llvm`. To generate debug information, attach `-g` as well. For example, The above IR example was generated by `clang -g -O0 -S -emit-llvm get_odd_even_diff.c -o get_odd_even_diff.ll`.

Function is defined as follows:

```llvm
1  define i32 @get_odd_even_diff(ptr noundef %0, i32 noundef
   %1) #0 !dbg !16 {
2    ...
3    %4 = alloca i32, align 4
4    ...
5    br label %8, !dbg !33
6
7  8:                                                ; preds = %27, %2
8    ...
9    br i1 %11, label %12, label %30, !dbg !38
10
11 20:                                               ; preds = %12
12   ...
13   %21 = load i32, ptr %5, align 4, !dbg !46
14   store i32 %22, ptr %5, align 4, !dbg !46
15   ...
```

- The `define` keyword is used to declare a function. The return type `i32` indicates that the function returns a 32-bit integer. The argument `(ptr noundef %0, i32 noundef %1)` means the function takes a pointer type and a 32-bit integer parameter named `%0` and `%1`, respectively. The `noundef` qualifier indicates that the argument must not be an undefined value.
- Line 3 allocates memory for a 32-bit integer (`i32`). At this point, it's not specified whether the integer is signed or unsigned. Later instructions, such as `srem i32 %a, %b` or `urem i32 %a, %b`, determine how it is treated.
- Line 5 transfers control to the basic block labeled `#8`. The next instruction to be executed is the first instruction of the basic block `#8`
- Line 9 performs a conditional branch based on the value of `%11`. If the condition evaluates to true, control is transferred to basic block `#12`; otherwise, it goes to basic block `#30`.
- Line 13 loads a `i32` type of value from pointer `%5` and stores it into `%21`
- Line 14 stores the value of `%22` into the memory location pointed to by `%5`. This stored value can be retrieved later using a load instruction.

As mentioned earlier, a basic block is a straight-line code sequence with no branches except at the end. In cases where the instruction sequence becomes long, it will break up into multiple basic blocks, even if there is no explicit control transfer at each step.

> **Remark 2.1** **Summary**
>
> All seven prepared topics operate on the IR. The instrumentation modules insert IR code at specific target locations. Using the IR binding library, we can create and manipulate global variables, functions, basic blocks, and instructions — including the ability to remove instructions.

Part II

Coverage

“How much of your code is tested?”

32% → 67% → 100%

3. Instrumentation and Runtime

3.1 Introduction to Coverage

What is the coverage [7]? Coverage is a measurement of which lines of code have been executed.

Have you ever used a coverage tool? Coverage is most commonly used in unit testing. Many developers aim to increase coverage to gain confidence in how thoroughly their code is being tested. For example, 100% coverage means that all lines of code are executed during testing, while 70% coverage indicates that 30% of the code has not been exercised by the tests.

Code coverage is one of the key topics covered in Software Engineering courses. Several global companies provide coverage tools that can be integrated with your CI pipelines or third-party tools like IDEs.

The goal of this chapter is to develop a **coverage instrumentation module** and a corresponding **runtime library** to measure function, branch, and line coverage. The implementation will be done in Rust, targeting programs written in C.

To guide our development, we'll model our tool after Hardhat's coverage tool [8]. Hardhat is a smart contract development framework that simplifies testing and deployment. Achieving high coverage in smart contracts is critical, as vulnerabilities can lead to real financial loss. From my experience, Hardhat Coverage is one of the most intuitive coverage tools I've used, which is why I chose it as a reference model.

3.2 Hardhat Coverage Example

First, let's take a look at a Hardhat Coverage example. The semantics of the example code are the same as those of the get_odd_even_diff function shown in this example.

Solidity

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.28;

contract TestContract {
    function getOddEvenDiff(int256[] memory arr) public pure returns
    (int256) {
```

```
        int256 evenCnt =0;
        int256 oddCnt =0;
        for (uint i=0; i<arr.length; i++) {
            if (arr[i] % 2 == 0) {
                evenCnt++;
            } else {
                oddCnt++;
            }
        }
        return oddCnt - evenCnt;
    }
}
```

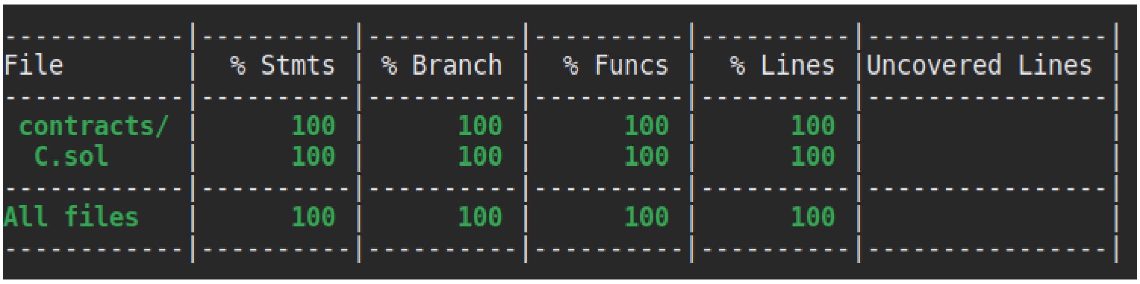
Let’s write a test for the contract.

```
const { expect } = require("chai");

describe("Test", function () {
  it("Test", async function () {
    const factory = await ethers.getContractFactory("TestContract");
    const contract = await factory.deploy();

    expect(await contract.getOddEvenDiff([1,2,3,4,5])).to.equal(1);
  })
});
```

Once you run `npx hardhat coverage`, the test code will be executed, and a summary will show which lines were hit and which were not, like this:



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
contracts/	100	100	100	100	
C.sol	100	100	100	100	
All files	100	100	100	100	

Figure 3.1: input = [1,2,3,4,5]

The input `[1,2,3,4,5]` covers all lines of code. Now, let’s try changing the input to `[1,3,5,7,9]` and `[2,4,6,8,10]`.

```
describe("Test", function () {
  it("Test", async function () {
    ...
```

```
    expect(await contract.getOddEvenDiff([1,3,5,7,9])).to.equal(5);
    ...
  })
});
```

The input `[1,3,5,7,9]` did not cover the line `10`.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
contracts/	100	50	100	85.71	
C.sol	100	50	100	85.71	10
All files	100	50	100	85.71	

Figure 3.2: input = [1,3,5,7,9]

The input `[2,4,6,8,10]` did not cover the line `12`.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
contracts/	100	50	100	85.71	
C.sol	100	50	100	85.71	12
All files	100	50	100	85.71	

Figure 3.3: input = [2,4,6,8,10]

3.3 Goal

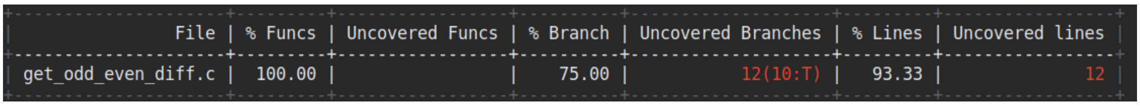
Finally implemented our coverage will report for the function get_odd_even_diff is as follows:

File	% Funcs	Uncovered Funcs	% Branch	Uncovered Branches	% Lines	Uncovered lines
get_odd_even_diff.c	100.00		100.00		100.00	

Figure 3.4: input = [1,2,3,4,5]

File	% Funcs	Uncovered Funcs	% Branch	Uncovered Branches	% Lines	Uncovered lines
get_odd_even_diff.c	100.00		75.00	10(12:F)	86.67	10,11

Figure 3.5: input = [1,3,5,7,9]



File	% Funcs	Uncovered Funcs	% Branch	Uncovered Branches	% Lines	Uncovered lines
get_odd_even_diff.c	100.00		75.00	12(10:T)	93.33	12

Figure 3.6: input = [2,4,6,8,10]

You can compile and run as fllows:

```Shell
> clang -g -O0 -S -emit-llvm get_odd_even_diff.c -o get_odd_even_diff.ll // compile origin program and produce IR
> ./instrument -i get_odd_even_diff.ll -o iget_odd_even_diff.ll -m coverage // instrument the compiled IR
> clang iget_odd_even_diff.ll -L. -lcoverage_runtime // compile the instrumented IR and produce binary
> LD_LIBRARY_PATH=./ ./a.out // run the binary
```

There are seven columns in our coverage report. The table below describes each reported item.

Reported item	Description
File	**File name**
% Funcs	**Degree of function coverage**
Uncovered Funcs	**Uncovered function lines**
% Branch	**Degree of branch coverage**
Uncovered Branch	**Uncovered branch lines**
% Lines	**Degree of line coverage**
Uncovered Lines	**Uncovered lines**

There are two conditional branches—one for the `for` loop and one for the `if-else` statement inside it. If the loop condition evaluates to `true`, the body executes; otherwise, control jumps outside the loop. This results in a total of four possible branches. For the inputs `[1, 3, 5, 7, 9]` and `[2, 4, 6, 8, 10]`, the for loop branches are fully covered, but either the `if` or `else` branch inside the loop is never taken. As a result, only 3 out of 4 branches are executed, leading to 75% branch coverage.

Remark 3.1 **Summary**

We introduced one of the most promising coverage tools, Hardhat Coverage. Our goal is to implement a similar UI and functionality. Additionally, we will provide detailed reports showing uncovered lines for each function, branch, and line.

3.4 How To Instrument

We will implement the initial instrumentation module to collect three types of coverage data: function, branch, and line coverage.

3.4.1 Instrument Strategy

First, we need to distinguish between functions, branches, and lines in the LLVM IR.

Most LLVM IR binding libraries provide utility functions to iterate over functions defined in a given IR module. Inkwell—the LLVM IR binding library [9] used by the book—also offers such functionality. This allows us to retrieve the definition line of each function by querying its debug metadata. To access this metadata, the IR must be generated with the `-g` compilation flag, which includes debugging information such as line numbers.

Second, for identifying branch lines, we must inspect each instruction to determine whether it is a branch instruction (***br ...***) or a switch instruction (***switch ...***). By iterating over the instructions within each basic block and checking their opcodes, we can accurately identify these control flow instructions.

Lastly, all instructions within a basic block is directly associated with lines.

3.4.1.1 Iterating IR file

First, the given IR file is parsed into a module structure managed by the binding library. Our analysis begins with this module. The module contains functions, each function contains basic blocks, and each basic block contains instructions—forming the hierarchy: "IR file ⟶ Module ⟶ Function ⟶ Basic Block ⟶ Instruction."

The following code demonstrates our common pattern for iterating over an IR module.

```rust
let funcs: Vec<_> = module.get_functions().collect();
for func in funcs {
    for basic_blk in func.get_basic_blocks() {
        for instr in basic_blk.get_instructions() {
            ...
        }
    }
}
```

3.4.1.2 Collecting line numbers

We need to collect all line numbers associated with functions, branches, and individual lines. The following function retrieves the line number for a given LLVM value, which may represent either a function or an instruction.

```rust
// instrument/src/llvm_intrinsic.rs
pub fn get_instr_loc<'ctx, T: AnyValue<'ctx>>(instr: &'ctx T) ->
(u32, u32) {
    unsafe {
        let line = LLVMGetDebugLocLine(instr.as_value_ref());
        let col = LLVMGetDebugLocColumn(instr.as_value_ref());
        (line, col)
    }
}
```

Here is how we collect line numbers.

Rust

```rust
// instrument/src/coverage.rs
let module_filename = cstr_to_str(module.get_source_file_name());
let mut brs_loc = vec![];
let mut lines_loc = BTreeSet::new();
let mut lines = BTreeSet::new();
for instr in basic_blk.get_instructions() {
    // Instrument within only given IR.
    // When we work with C++ IR, a lot of unknown files are collected together
    // We're interested for only our target C/C++ files, not other language system files.
    if let Some(instr_filename) = get_instr_filename(&instr) {
        if instr_filename != module_filename {
            continue;
        }
    }
    // Get line number of instructions and collect into a set.
    let (line, _) = get_instr_loc(&instr);
    lines.insert(line);
    lines_loc.insert(line);
    // Get line number of branch instruction and collect into a set.
    record_br(&mut brs_loc, &instr);
    record_switch(&mut brs_loc, &instr);
}
```

The handling of branch instructions is defined as follows:

Rust

```rust
// instrument/src/llvm_intrinsic.rs
pub fn record_br(brs_loc: &mut Vec<u32>, instr: &InstructionValue) {
    // if instruction is conditional branch
    if instr.is_conditional() {
        // if is `invoke` instruction
        if instr.get_opcode() == InstructionOpcode::Invoke {
            // get labels of normal and exception
            let (next_normal_label, exception_label) = (
                instr.get_operand(0).unwrap().right().unwrap(),
                instr.get_operand(1).unwrap().right().unwrap(),
            );
            if let Some(mut first_instr) = next_normal_label.get_first_instruction() {
                let loc = get_br_loc(&mut first_instr);
                brs_loc.push(loc);
            }
```

```
            if let Some(mut first_instr) = exception_label.get_first_instruction() {
                let loc = get_br_loc(&mut first_instr);
                brs_loc.push(loc);
            }
        }
        // if is `br` instruction
        if instr.get_opcode() == InstructionOpcode::Br {
            // get labels (basic blocks) of true and false branch
            let (tbr, fbr) = (
                instr.get_operand(2).unwrap().right().unwrap(),
                instr.get_operand(1).unwrap().right().unwrap(),
            );
            // collect line numbers
            if let Some(mut first_instr) = tbr.get_first_instruction() {
                let loc = get_br_loc(&mut first_instr);
                brs_loc.push(loc);
            }
            if let Some(mut first_instr) = fbr.get_first_instruction() {
                let loc = get_br_loc(&mut first_instr);
                brs_loc.push(loc);
            }
        }
    }
}
```

The invoke instruction typically appears in C++ code. Consider the example below:

```cpp
try {
  ... // normal label
} catch (...) {
  ... // exception label
}
```

If a runtime error occurs—such as an out-of-bounds access—within the code inside the `try` statement, control is transferred directly to the exception label. Otherwise, the exception label is never executed.

3.4.1.3 Build instrumentation

Once we have gathered the corresponding line numbers for a given IR, the final step is to instrument runtime functions that flag which lines have been executed.

A naive implementation is to instrument every IR line individually, such as:

```llvm
%1 = call ...
// call cov_hit_line(...)
%2 = load ...
// call cov_hit_line(...)
%3 = add ...
// call cov_hit_line(...)
```

The original IR consists of only three lines. In a naive instrumentation approach, a runtime function call is inserted for each IR line, resulting in a total of six lines—effectively doubling the code size.

Instead of instrumenting each line individually, we can optimize this by inserting a single runtime call per basic block. Since we already collect the line numbers for each basic block (as shown above), we declare a local array containing these line numbers.

As explained in Benefit of IR, all instructions within a basic block are guaranteed to execute before any control transfer (e.g., a branch) occurs. Therefore, we gather the line numbers of all instructions within a basic block, store them in a local array, and pass this array to a single runtime function call.

Below is the transformed version of the get_odd_even_diff function. The key changes include:

- Declaring a local array of line numbers
- Making a runtime function call with this array

```llvm
define i32 @get_odd_even_diff(ptr noundef %0, i32 noundef %1) #0 !dbg !9 {
  %__cov_hit_lines_arr = alloca [5 x i32], align 4 // [instrumented] array declaration
  store [5 x i32] [i32 0, i32 5, i32 6, i32 7, i32 8], ptr %__cov_hit_lines_arr, align 4 // [instrumented] intiailize array
  %__cov_hit_lines_arr__ptr = getelementptr [5 x i32], ptr %__cov_hit_lines_arr, i32 0, i32 0 // [instrumented] get pointer of the arr
  %3 = call i64 @__cov_hit_batch(ptr @for.c, ptr %__cov_hit_lines_arr__ptr, i64 5) // [instrumented] call runtime function call with the array pointer
  %4 = alloca ptr, align 8
  %5 = alloca i32, align 4
  %6 = alloca i32, align 4
  %7 = alloca i32, align 4
  %8 = alloca i32, align 4
  store ptr %0, ptr %4, align 8
  call void @llvm.dbg.declare(metadata ptr %4, metadata !15, metadata !DIExpression()), !dbg !16
  store i32 %1, ptr %5, align 4
  call void @llvm.dbg.declare(metadata ptr %5, metadata !17, metadata !DIExpression()), !dbg !18
  call void @llvm.dbg.declare(metadata ptr %6, metadata !19, metadata !DIExpression()), !dbg !20
```

```
16    store i32 0, ptr %6, align 4, !dbg !20
17    call void @llvm.dbg.declare(metadata ptr %7, metadata !21,
      metadata !DIExpression()), !dbg !22
18    store i32 0, ptr %7, align 4, !dbg !22
19    call void @llvm.dbg.declare(metadata ptr %8, metadata !23,
      metadata !DIExpression()), !dbg !25
20    store i32 0, ptr %8, align 4, !dbg !25
21    br label %9, !dbg !26
```

Lines 1 to 3 are instrumented—they declare an array, initialize it, and invoke a runtime function. No additional instrumentation is inserted within this basic block. In the next basic block, the same process is repeated. Therefore, we insert three lines of instrumentation code per basic block, rather than per individual line of code.

The following code snippets demonstrate how declarations and runtime function calls are constructed. The term ***build*** is commonly used in LLVM IR bindings and reflects the standard approach to IR instrumentation.

```Rust
// instrument/src/coverage.rs
let lines_len = lines.len(); => usize
if lines_len > 0 {
    let arr_ptr = build_i32_static_arr(
        context,
        builder,
        &lines.into_iter().collect(),
        COV_HIT_LINES_ARR,
        COV_HIT_LINES_ARR_PTR,
    )?;
    build_cov_hit_batch(
        context,
        module,
        builder,
        &filename_str_ptr.unwrap(),
        arr_ptr,
        lines_len,
    )?;
}
```

If there is a line within the current basic block, install the local array and the runtime function call using the `build_i32_static_arr()` and `build_cov_hit_batch()` build functions, respectively.

```Rust
// instrument/src/inkwell_intrinsic.rs
pub fn build_i32_static_arr<'ctx>(
    context: &'ctx Context,
    builder: &Builder<'ctx>,
```

```
    vals: &Vec<u32>,
    alloca_name: &str,
    gep_name: &str,
) -> Result<PointerValue<'ctx>> {
    let arr_typ = context.i32_type().array_type(vals.len().try_into()?); // Define array type
    let int_vals: Vec<_> = vals
        .iter()
        .map(|v| context.i32_type().const_int(*v as u64, false))
        .collect(); // Gather element values (line numbers)
    // Build(instrument) array
    let arr_val = context.i32_type().const_array(&int_vals);
    let arr_alloca = builder.build_alloca(arr_typ, alloca_name)?;
    builder.build_store(arr_alloca, arr_val)?;

    // Build declaration of array pointer
    // Get a pointer to the first element
    let zero = context.i32_type().const_int(0, false);
    let array_ptr = unsafe {
        builder.build_gep(
            arr_typ,
            arr_alloca,
            &[zero, zero], // Get pointer to the first element
            gep_name,
        )
    }?;
    Ok(array_ptr)
}
```

To build a function call, we should create a function first.

```rust
// instrument/src/inkwell_intrinsic.rs
fn get_cov_hit_batch_func<'ctx>(
    context: &'ctx Context,
    module: &Module<'ctx>,
) -> FunctionValue<'ctx> {
    // If the function had been built before, get the function value
    // Otherwise, build it.
    match get_func(module, COV_HIT_BATCH) {
        Some(func) => func,
        None => {
```

```
            let record_func_cov_lines_typ = context.i64_type().fn_type(
                &[
                    context.ptr_type(AddressSpace::default()).into(),
                    context.ptr_type(AddressSpace::default()).into(),
                    context.i64_type().into(),
                ],
                false,
            );
            module.add_function(COV_HIT_BATCH, record_func_cov_lines_typ, None)
        }
    }
}

// instrument/src/inkwell_intrinsic.rs
pub fn build_cov_hit_batch<'ctx>(
    context: &'ctx Context,
    module: &Module<'ctx>,
    builder: &Builder<'ctx>,
    filename_str_ptr: &GlobalValue,
    arr_ptr: PointerValue,
    arr_length: usize,
) -> Result<()> {
    let record_func_cov_lines = get_cov_hit_batch_func(context, module);
    // Call coverage runtime function with three parameters, file name, array pointer, and size of array.
    builder.build_call(
        record_func_cov_lines,
        &[
            filename_str_ptr.as_pointer_value().into(),
            arr_ptr.into(),
            convert_to_int_val(context, arr_length.try_into()?).into(),
        ],
        "",
    )?;
    Ok(())
}
```

The process of creating a runtime function signature (function declaration) is extensively used in other modules as well.

So far, we do not know how many lines of code are covered. To obtain this information, the runtime must know both how many lines of code are executable and which lines have been executed. Currently, we have only handled the latter.

To calculate the coverage percentage, the final step is to build a runtime initialization function. This function collects all executable lines related to functions, branches, and lines within the IR module, not the basic block level. Later, this information is used to determine how many lines have been covered by comparing it with all executable lines. This build function is executed at the final phase of the instrumentation process.

```rust
// instrument/src/coverage.rs
let init_last_instr = get_cov_init_last_instr(module);
builder.position_before(&init_last_instr);
build_src_mapping_call(
    context,
    module,
    builder,
    &filename_str_ptr.unwrap(),
    &funcs_loc,
    &brs_loc,
    &lines_loc,
)?;
```

Record the total number of code lines for functions, branches, and lines. First, a local array is constructed; then, this array is passed to a runtime function that initializes and records all the code line information.

```rust
// instrument/src/inkwell_intrinsic.rs
pub fn build_src_mapping_call<'ctx>(
    context: &'ctx Context,
    module: &Module<'ctx>,
    builder: &Builder<'ctx>,
    filename_str_ptr: &GlobalValue,
    funcs_loc: &BTreeSet<u32>,
    brs_loc: &Vec<u32>,
    lines_loc: &BTreeSet<u32>,
) -> Result<()> {
    let funcs_loc_vec: Vec<_> =
    funcs_loc.clone().into_iter().collect();
    let brs_loc_vec: Vec<_> = brs_loc.clone().into_iter().collect();
    let lines_loc_vec: Vec<_> =
    lines_loc.clone().into_iter().collect();

    let func_lines_ptr = build_i32_static_arr(
```

```
        context,
        builder,
        &funcs_loc_vec,
        COV_SRC_MAPPING_FUNC_LINES,
        COV_SRC_MAPPING_FUNC_LINES_PTR,
    )?;
    let brs_lines_ptr = build_i32_static_arr(
        context,
        builder,
        &brs_loc_vec,
        COV_SRC_MAPPING_BRS_LINES,
        COV_SRC_MAPPING_BRS_LINES_PTR,
    )?;
    let lines_lines_ptr = build_i32_static_arr(
        context,
        builder,
        &lines_loc_vec,
        COV_SRC_MAPPING_LINES_LINES,
        COV_SRC_MAPPING_LINES_LINES_PTR,
    )?;

    builder.build_call(
        get_src_mapping_func(context, module),
        &[
            filename_str_ptr.as_pointer_value().into(),
            func_lines_ptr.into(),
            convert_to_int_val(context, funcs_loc_vec.len().try_into()?).into(),
            brs_lines_ptr.into(),
            convert_to_int_val(context, brs_loc_vec.len().try_into()?).into(),
            lines_lines_ptr.into(),
            convert_to_int_val(context, lines_loc_vec.len().try_into()?).into(),
        ],
        "",
    )?;
    Ok(())
}
```

A key point is that the source mapping function is created within a ***constructor*** [10] function. In C/C++, a constructor function runs automatically when the process loads—before the `main()` function executes. If the given IR already defines constructor functions, it's not an issue because multiple constructors can coexist and are prioritized based on a specified priority parameter. Our construction function works regardless of when it is executed within the set of construction functions.

3.4.1.4 Coverage Runtime

We've instrumented a runtime function call for each basic block, and now it's time to define the runtime function itself. Before diving into the implementation, let's take a moment to recap the bigger picture of what we're trying to achieve.

Instrumentation

Collect line numbers and build runtime function calls at compile time

Runtime

Runtime function records which lines have been executed and output a report of coverage when a program exits

There are three exposed runtime functions.

- `__cov_init()`: Called at the constructor function
  1. Register report function to be called at program exit via `libc::atexit()`
  2. Initialize internal coverage structure
- `__cov_mapping_src(...)`: Called at the constructor function
  - Record all code lines
- `__cov_hit_batch(...)`: Called per function, branch, and instructions
  - Record which lines are hit

Rust

```rust
// coverage_runtime/src/coverate_runtime.rs
#[no_mangle]
pub extern "C" fn __cov_mapping_src(
    file_ptr: *const libc::c_char,
    funcs_ptr: *const u32,
    funcs_length: usize,
    brs_ptr: *const u32,
    brs_length: usize,
    lines_ptr: *const u32,
    lines_length: usize,
) {
    if file_ptr.is_null() || funcs_ptr.is_null() || brs_ptr.is_null() || lines_ptr.is_null() {
        return;
    }

    let filename = cstr_to_string(file_ptr);
    let (funcs, brs, lines) = unsafe {
        (
            std::slice::from_raw_parts(funcs_ptr, funcs_length),
```

```
            std::slice::from_raw_parts(brs_ptr, brs_length),
            std::slice::from_raw_parts(lines_ptr, lines_length),
        )
    };
    let src_map = SourceMapping {
        lines: lines.to_vec(),
        brs: brs.to_vec(),
        funcs: funcs.to_vec(),
    };
    COVERAGE_STATE
        .write()
        .unwrap()
        .source_map
        .lock()
        .unwrap()
        .insert(filename, src_map);
}

#[no_mangle]
pub extern "C" fn __cov_hit_batch(
    file_ptr: *const libc::c_char,
    lines_ptr: *const u32,
    length: usize,
) {
    if lines_ptr.is_null() {
        return;
    }

    // convert raw C pointer into rust slice
    let lines = unsafe { std::slice::from_raw_parts(lines_ptr, length) };
    for line in lines {
        __cov_record(file_ptr, *line);
    }
}

/// Records a hit at the given source location
#[no_mangle]
pub extern "C" fn __cov_record(file_ptr: *const libc::c_char, line: u32) -> usize {
    // Early return if coverage is disabled
    if COVERAGE_STATE
```

```
        .read()
        .unwrap()
        .enabled
        .load(Ordering::Relaxed)
        == 0
    {
        return 0;
    }

    let file = cstr_to_string(file_ptr);
    let loc = LineMapping { file, line };

    // Get or create counter for this location
    let lines_idx = {
        let state = COVERAGE_STATE.write().unwrap();
        let mut loc_map = state.location_map.lock().unwrap();
        if let Some(&idx) = loc_map.get(&loc) {
            idx
        } else {
            let idx = state.lines.len();
            loc_map.insert(loc, idx);
            idx
        }
    };
    let mut state = COVERAGE_STATE.write().unwrap();
    if lines_idx >= state.lines.len() {
        // Insert a new index
        state.lines.push(AtomicUsize::new(0));
    }
    // Increment the lines
    state.lines[lines_idx].fetch_add(1, Ordering::Relaxed)
}
```

There are several ways to collect a report over the program's lifetime. In this book, we use the `atexit()` function [11] to register a report-generating function, which ensures that the coverage report is automatically generated when the program terminates.

The final function, `__cov_hit_batch()`, records hit information into the global state. This global state primarily manages the following two sets:

- `state.source_map`: Stores all code lines at runtime intiailization
- `state.location_map`: Stores hit line per instructions

The `#[no_mangle]` annotation is necessary for our runtime functions to be callable from the instrumented program. Since we're writing the shared library in Rust, any functions we want to export must not be mangled.

If you've worked with C++, you may have encountered name mangling [12]. C++ supports function overloading, allowing multiple functions with the same name but different parameter types. To distinguish these during compilation, the compiler encodes type information into function names—a process known as name mangling. As a result, each overloaded version gets a unique symbol name.

In our case, the instrumented program must call the runtime function using its exact name. Therefore, name mangling must be disabled to ensure the function is exposed with the correct symbol name.

The report function computes coverage metrics at the function, branch, and line levels. Once this functionality is in place, we can improve the output presentation by formatting it into a table and adding colors for readability.

3.4.1.5 Integration with Test Environment (gtest)

The most common use case for coverage information is its integration into unit testing environments. In this example, we'll use `gtest`, a highly popular C/C++ testing framework.

Navigate to the `unit-test` directory. Within this example, we'll simply demonstrate testing a basic math function:

```c
#include <math.h>

int add(int a, int b) {
    int v1 = a;
    int v2 = b;
    return v1 + v2;
}

int mul(int a, int b) {
    int v1 = a;
    int v2 = b;
    return v1 * v2;
}

int unused(int a, int b) {
    int v1 = a;
    int v2 = b;
    return v1 + v2;
}
```

math.c contains simple `add`, `mul`, and `unused` functions. The `unused` function will be detected as untouched in the coverage results.

This is the unit test file that tests math coverage:

```c
#include <gtest/gtest.h>

extern "C" {
```

```
    #include "math.h"
}

TEST(MathTest, Add) {
    EXPECT_EQ(add(2, 3), 5);
    EXPECT_EQ(mul(2, 3), 6);
}

int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

The test file exclusively invokes the `add` and `mul` functions. Consequently, the `unused()` function should be reported as uncovered.

Let's run

```
> just build
> just run
LD_LIBRARY_PATH=../../bin/debug ./test_math
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from MathTest
[ RUN      ] MathTest.Add
[       OK ] MathTest.Add (0 ms)
[----------] 1 test from MathTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (0 ms total)
[  PASSED  ] 1 test.
+--------+---------+-----------------+----------+--------------------+---------+-----------------+
|   File | % Funcs | Uncovered Funcs | % Branch | Uncovered Branches | % Lines | Uncovered lines |
+--------+---------+-----------------+----------+--------------------+---------+-----------------+
| math.c |   66.67 |              15 |      NaN | |   69.23 |     15,16,17,18 |
+--------+---------+-----------------+----------+--------------------+---------+-----------------+
```

As we can see, the function coverage is not 100%. All lines within the `unused()` function's body are uncovered, specifically reported as lines 15, 16, 17, and 18.

3.4.1.6 Summary

We've implemented the first instrumentation module and its corresponding runtime library. The instrumentation module operates at compile time, generating coverage-related information that calls the runtime library. The runtime library, in turn, manages and reports this coverage data during program execution. In essence, the instrumentation prepares the data we care about, and the runtime library processes it at runtime. This combination of compile-time and runtime techniques is a recurring pattern in the chapters that follow.

For the complete coverage code, refer to the `instrument` and `coverage_runtime` directories.

Coverage serves as a warm-up to help you become familiar with the instrumentation-runtime workflow. In the next chapter, we'll apply the same principle to build an out-of-bounds access detector known as AddressSanitizer (ASAN).

You can check out provided test cases in `coverage/tools/tests/inputs/coverage`.

3.4.1.7 Homework

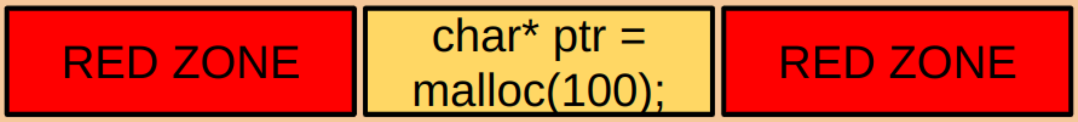
Exercise 3.1 We suggest the following topics for further study and exploration:

- **Topic #1: Reducing Runtime Function Calls with Dominance Graphs**
  - The current implementation places runtime function calls at the basic block level. However, this overhead can be further reduced by using a dominance graph [13]. By constructing a dominance relationship between basic blocks, we can identify opportunities to eliminate redundant instrumentation. For example, if the execution path is guaranteed to follow BB1 ⟶ BB3 ⟶ BB7, then placing the runtime function call in BB1 alone is sufficient. This allows us to eliminate the calls in BB3 and BB7, reducing runtime overhead.
- **Topic #2: Further information**
  - The runtime state variable `state.lines` tracks the number of executions per line. This information can be leveraged to identify hotspots or cold spots in the code. For example, lines with unusually high or low hit counts may indicate performance-critical code paths or dead code, respectively. Further analysis of this data can provide deeper insights into code behavior and optimization opportunities.

Part III

Buffer Overrun: Address Sanitizer

“Hardening the memory usage to improve safety”



4. Introduction to Out-of-bound Access Bug

4.1 Buffer Overrun

Lists are among the most dominant data structures in modern programming languages. Many languages abstract raw arrays to provide additional functionality, such as dynamic sizing and built-in length tracking. It's hard to imagine programming without some form of list structure.

However, arrays (and lists) come with a well-known pitfall: out-of-bounds (OOB) access bugs. These occur when a program tries to access memory outside the valid range of an array or buffer.

Chances are, many readers have encountered OOB bugs at some point. Consider the following C code, which demonstrates an OOB access:

```c
1 #include <stdio.h>
2
3 int main() {
4         int arr[3] = {1,2,3};
5         printf("%d\n", arr[2]); // 3
6         printf("%d\n", arr[3]); // ?
7 }
```

Line 6 accesses an index out of bounds. The `arr` occupies 12 bytes, allowing valid access only within the range of indices 0 to 2. Therefore, `arr[3]` accesses memory outside the defined bounds, resulting in undefined behavior. The returned value is unpredictable—some compilers may return zero, while others may produce garbage values. Regardless of the outcome, this is clearly a bug.

A more serious issue is when the program continues executing without reporting the bug. The invalid value may propagate through subsequent instructions, leading to undefined behavior (UB) [14]. UB makes debugging significantly harder.

Rather than allowing the program to continue silently, modern compilers and runtimes often instrument checks to detect out-of-bounds (OOB) accesses and crash the program with a helpful call stack trace. As a result, modern programming languages are generally better equipped to detect and report OOB bugs compared to traditional compilers like GCC or Clang.

Let's take a look the Golang code with the same sementic.

```go
package main

import "fmt"

func main() {
    arr := []int{1, 2, 3}
    fmt.Println(arr[2]) // 3
    fmt.Println(arr[3]) // crash
}
```

Go explicitly reports OOB errors and prevents the program from continuing execution. This behavior helps developers quickly identify and fix such bugs. When an OOB access occurs at runtime, Go outputs an error message as fllows:

```shell
> go run oob.go
3
panic: runtime error: index out of range [3] with length 3

goroutine 1 [running]:
main.main()
        .../oob.go:8 +0x87
exit status 2
```

Now, let's look at a Rust example.

```rust
fn main() {
    let arr = vec![1,2,3];
    println!("{}", arr[2]);
    println!("{}", arr[3]); // crash
}
```

Compiling and running the code will produce the following output:

```shell
> rustc oob.rs && ./oob
3

thread 'main' panicked at oob.rs:4:23:
index out of bounds: the len is 3 but the index is 3
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

4.1.1 Goal

As we observed the discrepancy between traditional compilers (*Gcc* and *Clang*) and modern compilers (*Go* and *Rust*), the latter have recognized common error patterns—such as OOB access—and made significant efforts to create a more robust programming environment through both instrumentation and runtime mechanisms. In this chapter, we will implement an OOB detector using a custom instrumentation module and a runtime library. **A naive C program will be compiled to LLVM IR, then instrumented using our custom ASAN instrumentation. If an OOB access occurs during execution, our ASAN runtime will detect and report it.**

For the OOB code snipeet, let's compile the C program and instrument, and attach runtime library.

```shell
> clang -g -O0 -S -emit-llvm asan-example.c -o iasan-example.ll // compile origin program and produce IR
> ./instrument -i iasan-example.ll -o iasan-example.ll -m asan // instrument the generated IR
> clang iasan-example.ll -L. -lasan_runtime // compile the instrumented IR and produce binary
```

Our OOB detector will report as follows:

```shell
> LD_LIBRARY_PATH=./ ./a.out // libasan_runtime.so should be located in the current directory. Otherwise you should specify the exact path where the library places.
3
[ASAN] invalid memory access detected at asan-example.c: 0x7ffcfc3fbf20
   0: asan_runtime::asan_runtime::report_asan_violated::{{closure}}
             at .../asan_runtime/src/asan_runtime.rs:46:18
   1: std::thread::local::LocalKey<T>::try_with
             at .../.rustup/toolchains/stable-x86_64-unknown-linux-gnu/lib/rustlib/src/rust/library/std/src/thread/local.rs:308:12
   2: std::thread::local::LocalKey<T>::with
             at .../.rustup/toolchains/stable-x86_64-unknown-linux-gnu/lib/rustlib/src/rust/library/std/src/thread/local.rs:272:9
   3: asan_runtime::asan_runtime::report_asan_violated
             at .../asan_runtime/src/asan_runtime.rs:44:5
   4: __asan_mem_check
             at .../coverage/asan_runtime/src/asan_runtime.rs:30:9
   5: main
             at ./asan-example.c:6:22
   6: __libc_start_call_main
   7: __libc_start_main_alias_2
   8: _start
```

Running the instrumented program with runtime library

The OOB detector reports a stack trace and identifies the location where the OOB access occurred. Lines 14 and 15 pinpoint the exact location of the issue. Specifically, in the `main` function, line 6 is identified as the point of the OOB access. Our detector is capable of handling both heap and stack buffer overflows.

4.1.2 Reference Paper

In this chapter, we refer to a seminal paper that is considered a foundational work in this research area: ***"AddressSanitizer: A Fast Address Sanity Checker"*** (ASan) [15]. AddressSanitizer is a tool designed to detect out-of-bounds (OOB) memory errors. The paper primarily focuses on five categories of memory-related bugs. (Use After Free (UAF) [16], Buffer Overflow (BOF) [17], Uninitialized Read [16], Double Free [18], Null Dereference[19]).

ASan is currently built into the Clang compiler toolchain [20] and can be used in any C project.

Bug Category	Description
Use-After-Free (UAF)	**Using memory after it's freed**
Buffer Overflow	**Accessing memory outside valid bound**
Uninitializd Read	**Reading memory without initialization**
Double Free	**Freeing memory more than once**
Null Dereference	**Dereferencing a null pointer**

In this book, we implement only two categories of bugs: UAF and BOF.

4.2 Idea

In the following code, the variable `arr` occupies 12 bytes of stack memory. To detect memory violations, shadow memory is introduced. Shadow memory tracks the allocation and deallocation status of each memory region, allowing the system to monitor when and where memory is allocated or freed.

```c
int arr[3] = {1,2,3};
arr[2] // 3
arr[3] // OOB
```

4.2.1 Red Zone

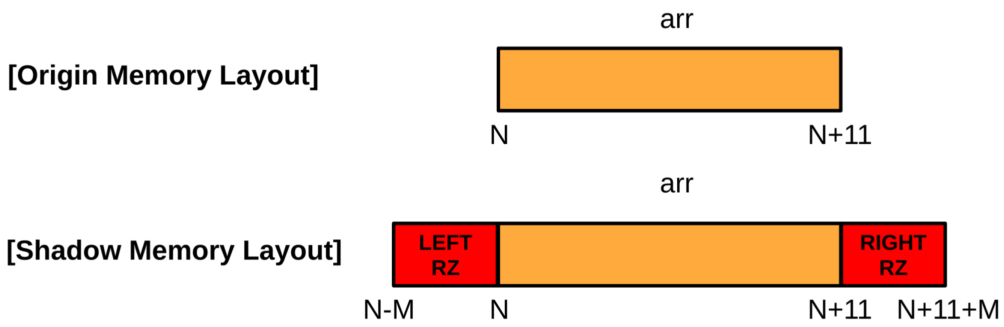


Figure 4.1: Shadow Memory Layout: Adds redzones to both the left and right of the original memory layout.

The figure illustrates two memory layouts: the original memory layout of the variable `arr` and its corresponding shadow memory layout. In the original layout, 12 bytes are allocated exactly as requested—nothing more. In contrast, the shadow memory layout includes additional space called "***redzones***" placed before and after the allocated region to detect OOB access. Assuming each redzone is 32 bytes, a total of 64 extra bytes are allocated—tagged as the left and right redzones, respectively.

Accessing index 3 is reliably detected because it falls within the right redzone, which is tracked by the shadow memory. However, if the access goes beyond the redzone boundaries, it may go undetected. This introduces a trade-off: increasing the redzone size improves the likelihood of catching out-of-bounds (OOB) accesses but also increases memory overhead. Conversely, reducing the redzone size optimizes memory usage but may lower the detection rate. Typically, a redzone size of 32 bytes is used as a balanced default. See the two cases illustration for examples.

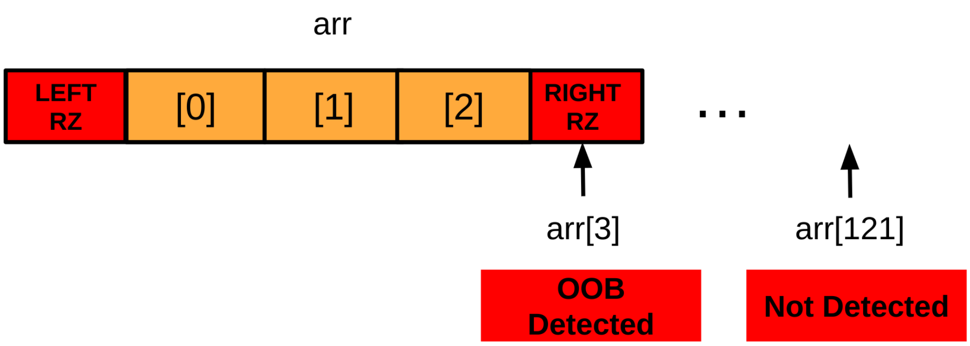


Figure 4.2: Cases: When OOB access is detected vs. not detected

4.2.2 Allocator Overloading

To automatically manage redzones, the ASan runtime must intercept memory allocation and deallocation requests. Here's how it works:

- **Allocation** (e.g., 12-byte request):

  - How many bytes are allocated?
    - A total of 76 bytes is allocated
      - 32 bytes for the left redzone
      - 12 bytes for the usable region
      - 32 bytes for the right redzone
      - Although the redzones are not used for program data, they must occupy memory to prevent the OS from reusing those regions.
- **Which address is returned?**
  - The allocator returns the address pointing to the start of the usable region (offset 32).
    - Address 0–31: Left redzone
    - Address 32–43: Usable memory
    - Address 44–75: Right redzone
    - The redzones are never returned to the program; they are not exposed through any legal pointer.
- **What happens in the shadow memory?**
  1. Shadow memory marks the left and right redzones with a special magic value to indicate that they are poisoned.
  2. During program execution, instrumented code checks the shadow memory to determine whether a memory access touches a poisoned (redzone) area.
- **Deallocation**
  1. The allocator frees the entire 76-byte block, including the redzones and the usable region.
  2. Shadow memory is updated to mark the entire region as freed.
- **Reallocation**
  1. Mark the entire existing region (including redzones) as freed in the shadow memory.
  2. Allocate a new block with redzones surrounding the requested size.
  3. Copy the contents of the original usable region into the new usable region.
  4. Deallocate the previous full block, including redzones.
  5. Return a pointer to the new usable region.

ASan allocates additional memory—called redzones—whenever a memory allocation request is made during program execution. It also maintains a separate region known as shadow memory, which is managed exclusively by the ASan runtime. This shadow memory is reserved when the ASan runtime library is first loaded.

In the referenced paper, shadow memory occupies one-eighth of the virtual address space. Allocating such a large memory region is not problematic due to how OSes manage memory. Specifically, virtual memory is reserved upfront, but physical memory is only allocated on demand, page by page, when the memory is actually accessed or modified. This lazy allocation ensures efficiency while allowing ASan to maintain a large shadow memory space without consuming excessive physical resources.

4.2.3 Shadow Memory Translation

A given memory address is mapped to a shadow memory address using the formula `(Address >> 3) + Offset` (which is equivalent to `Address / 8 + Offset`). This means that one byte in shadow memory represents 8 bytes of actual memory.

Because of this 1:8 mapping ratio, the paper allocates one-eighth of the virtual address space for shadow memory—just enough to cover the entire range of possible memory addresses in the program.

The `Offset` represents the starting address of the shadow memory. A simplified interpretation of the mapping is: `SHADOW_MEMORY[Addr >> 3]`.

There's a subtle issue when marking shadow memory. As shown in the figure below, both `arr[0]` and `arr[1]` map to index 0 in the shadow memory. Similarly, `arr[2]` maps to index 2 in the shadow memory. Because of the 1:8 mapping ratio, the next 4 bytes following `arr[2]` (including `arr[3]`) are also covered by the same shadow memory byte at index 2. As a result, OOB access to `arr[3]` may go undetected.

To address this, boundary poisoning is introduced, allowing finer-grained marking of the exact boundaries of valid memory and detecting partial OOB accesses.

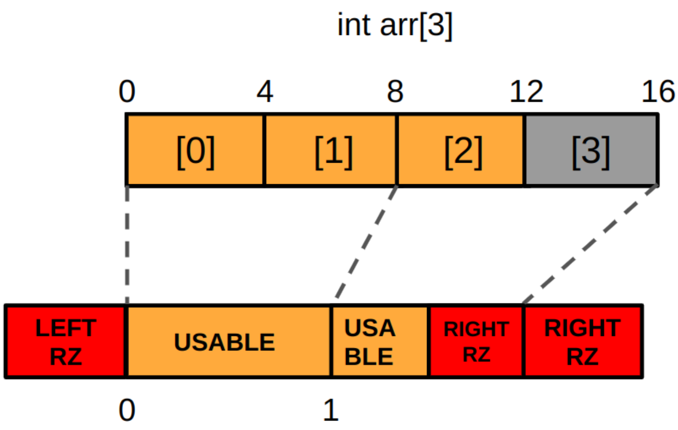


Figure 4.3: Boundary poisoning

The boundary within a shadow memory byte is determined using a bitwise AND operation: `address & 0x07`. For example, if the last address of the variable `arr` is 12, the calculation `12 & 0x07` results in 4. This means that only the first 4 bytes within that 8-byte block are valid. To reflect this, the value `0x04` is written into index 1 of the shadow memory. This indicates that only 4 bytes are part of the usable region. When the variable `arr` is accessed, the instrumented code (explained later) performs a check based on this shadow memory value to determine if the access is valid.

```
ShadowAddr = (Addr >> 3) + Offset;
k = *ShadowAddr;
if (k != 0 && ((Addr & 7) + AccessSize > k))
    ReportAndCrash(Addr);
```

Consider an access to `arr[3]`. In this case:

- `ShadowAddr = 12 >> 3 = 1`
- `k = 4` (the value stored in shadow memory at index 1, indicating that only the first 4 bytes are valid)
- `AccessSize = 4` (since the array holds integers, each access is 4 bytes)
- At runtime, `if` condition evaluates true: `if (4 != 0 && (4 + 4 > 4))`
- Correctly triggers an OOB report for `arr[3]`

Now, consider a valid access to `arr[2]`:

- `ShadowAddr = 8 >> 3 = 1`
- `k = 4`
- `AccessSize = 4`
- `if (4 != 0 && (4 + 0 > 4)) → false`
- So the access is considered safe, and no error is reported.

> **Remark 4.1** — **Access range is important**
>
> An important observation is that if the `AccessSize` is 5, the condition evaluates to true: `if (4 != 0 && (5 + 0 > 4))` This correctly triggers an OOB report, since accessing 5 bytes starting at address 8 would reach address 13—beyond the valid region—which is not part of the usable memory.

4.2.4 Implementation

ASan is implemented with two components, instrumentation and runtime.

- **Instrumentation Module**: *Instrument runtime call for all memory access operations*
- **Runtime Library**: *Assert if the access is legal*

4.2.4.1 ASan Instrumentation

In LLVM IR, memory access operations are represented by two instructions: `store` and `load`.

Arrays can be allocated either on the **stack** or on the **heap**. The key difference lies in when their sizes are determined: stack-allocated arrays have a *statically known size* at compile time, whereas heap-allocated arrays are created with a *dynamically determined size* at runtime.

For heap allocations, memory allocation and deallocation are performed through OS-level requests (e.g., `malloc()` / `free()`), which our runtime library intercepts to manage redzones and shadow memory.

In contrast, for statically sized stack arrays, the compiler knows the size at compile time. During instrumentation, the module will insert calls to our runtime to initialize the corresponding shadow memory for the static array.

Here is the main logic of ASan instrumentation module.

```rust
#[derive(Default)]
pub struct ASANModule {}

impl InstrumentModule for ASANModule {
    fn instrument<'ctx>(
        &self,
        context: &'ctx Context,
        module: &Module<'ctx>,
        builder: &Builder<'ctx>,
    ) -> Result<()> {
        let mut filename_str_ptr = None;
        let funcs: Vec<_> = module.get_functions().collect();
```

```
        for func in funcs {
            // Skip funcs without bodies or those we've added
            if can_skip_instrument(&func) {
                continue;
            }
            set_filename(module, builder, &mut filename_str_ptr, &func)?;
            let mut replaced_alloca = HashMap::new();
            let mut instrumented_blks = HashSet::new();
            for basic_blk in func.get_basic_blocks() {
                if instrumented_blks.contains(&basic_blk) {
                    continue;
                }
                for instr in basic_blk.get_instructions() {
                    match instr.get_opcode() {
                        // install asan check
                        InstructionOpcode::Load => {
                            handle_load(context, module, builder, filename_str_ptr, &instr)?;
                        }
                        InstructionOpcode::Store => {
                            handle_store(context, module, builder, filename_str_ptr, &instr)?;
                        }
                        InstructionOpcode::Alloca => {
                            handle_alloca(context, module, builder, &mut replaced_alloca, &instr)?;
                        }
                        // Replace all uses of origin static object with newly allocated object's pointer
                        InstructionOpcode::Call => {
                            handle_call(context, builder, &replaced_alloca, &instr)?;
                        }
                        InstructionOpcode::GetElementPtr => {
                            handle_gep(context, builder, &replaced_alloca, &instr)?;
                        }
                        _ => {}
                    }
                }
                instrumented_blks.insert(basic_blk);
            }
```

```
        }
        // Verify instrumented IRs
        module_verify(module)
    }
}
```

The instrumentation targets five key LLVM instructions: load, store, alloca, call, and getelementptr. Let's begin by examining the load and store instructions. As their names suggest, both are directly related to memory access operations. We'll start with the `handle_load()` function to understand how load instructions are handled.

```rust
// instrument/src/inkwell_intrinsic.rs
pub fn get_ptr_operand<'ctx>(instr: &InstructionValue<'ctx>, idx: u32) -> PointerValue<'ctx> {
    instr
        .get_operand(idx)
        .unwrap()
        .left()
        .unwrap()
        .into_pointer_value()
}

// instrument/src/asan.rs
fn build_memcheck<'ctx>(
    context: &'ctx Context,
    module: &Module<'ctx>,
    builder: &Builder<'ctx>,
    filename_str_ptr: Option<GlobalValue<'ctx>>,
    instr: &InstructionValue<'ctx>,
    ptr: PointerValue<'ctx>,
    access_size: IntValue<'ctx>,
) -> Result<()> {
    // the index value must be integer type when using it as array index (RHS) value (e.g., int val = arr[idx] + 1)
    builder.position_before(&instr);
    build_asan_mem_check(
        context,
        module,
        builder,
        &filename_str_ptr.unwrap(),
        ptr,
        access_size,
    )?;
```

```
    Ok(())
}

// instrument/src/asan.rs
fn handle_load<'ctx>(
    context: &'ctx Context,
    module: &Module<'ctx>,
    builder: &Builder<'ctx>,
    filename_str_ptr: Option<GlobalValue<'ctx>>,
    instr: &InstructionValue<'ctx>,
) -> Result<()> {
    let ptr = get_ptr_operand(&instr, 0);
    if instr.get_type().is_int_type() {
        let access_size = instr.get_type().into_int_type().size_of();
        build_memcheck(
            context,
            module,
            builder,
            filename_str_ptr,
            &instr,
            ptr,
            access_size,
        )?;
    }
    Ok(())
}
```

From the load instruction, we can extract the pointer operand that specifies the memory location to read from. This is done using the `get_ptr_operand()` function, which retrieves the operand from the instruction. In a `load` instruction, the first operand is always the pointer to the data being accessed.

The access size can be determined by querying the type of the loaded value, which is straightforward for statically sized arrays since their types are known at compile time. For simplicity, we handle only integer arrays. However, readers can extend the implementation to support more general types.

Finally, we instrument a call to the runtime to check whether the memory access falls within a valid range. In summary, from a `load` instruction, we extract both the pointer address and the access size and pass them to the runtime call arguments.

```rust
// instrument/src/inkwell_intrinsic.rs
fn get_asan_mem_check_func<'ctx>(
    context: &'ctx Context,
    module: &Module<'ctx>,
```

```
) -> FunctionValue<'ctx> {
    match get_func(module, ASAN_MEM_CHECK) {
        Some(func) => func,
        None => {
            let asan_mem_check_func_typ = context.void_type().fn_type(
                &[
                  context.ptr_type(AddressSpace::default()).into(),
                  context.ptr_type(AddressSpace::default()).into(),
                  context.i64_type().into(),
                ],
                false,
            );
            module.add_function(ASAN_MEM_CHECK, asan_mem_check_func_typ, None)
        }
    }
}

// instrument/src/inkwell_intrinsic.rs
pub fn build_asan_mem_check<'ctx>(
    context: &'ctx Context,
    module: &Module<'ctx>,
    builder: &Builder<'ctx>,
    filename_str_ptr: &GlobalValue,
    ptr: PointerValue,
    access_size: IntValue<'ctx>,
) -> Result<()> {
    let asan_mem_check = get_asan_mem_check_func(context, module);
    builder.build_call(
        asan_mem_check,
        &[
            filename_str_ptr.as_pointer_value().into(),
            ptr.into(),
            access_size.into(),
        ],
        "",
    )?;
    Ok(())
}
```

Building the runtime call for memory access assertions is straightforward. As with the Coverage module, the first step is to define the signature of the runtime function. This function takes three parameters: a pointer to the file name, the pointer to the array being accessed, and the size of the access.

Handling the `store` instruction follows a similar process, with the main difference being how the parameters are retrieved from the instruction.

```rust
// instrument/src/inkwell_intrinsic.rs
fn handle_store<'ctx>(
    context: &'ctx Context,
    module: &Module<'ctx>,
    builder: &Builder<'ctx>,
    filename_str_ptr: Option<GlobalValue<'ctx>>,
    instr: &InstructionValue<'ctx>,
) -> Result<()> {
    let offset = instr.get_operand(0).unwrap().left().unwrap();
    let ptr = get_ptr_operand(&instr, 1);

    // pointer value is used as LHS in `store` instruction. (e.g.,
    arr[idx] = value)
    // any type of access size is valid (i.e., `= value`)
    let access_size = offset.get_type().size_of().unwrap();
    build_memcheck(
        context,
        module,
        builder,
        filename_str_ptr,
        &instr,
        ptr,
        access_size,
    )?;
    Ok(())
}
```

The remaining instructions to handle are `alloca`, `call`, and `getelementptr`. Let's start by exploring the `alloca` instruction.

A statically allocated array—for example, `int array[3] = {1, 2, 3}`—is represented in LLVM IR as `%1 = alloca [3 x i32], align 4`. To support redzones, our instrumentation module needs to replace this instruction with a new allocation that includes additional space for redzones while preserving the original semantics of the initialization. After instrumentation, the modified instruction becomes `%1 = alloca [19 x i32], align 4`, which allocates a total of 76 bytes—comprising the original 12 bytes and an additional 64 bytes for the redzones.

After performing the replacement, we insert a call to a runtime function to initialize the corresponding shadow memory.

Here's how this can be implemented:

```rust
// instrument/src/inkwell_intrinsic.rs
fn handle_alloca<'ctx>(
    context: &'ctx Context,
    module: &Module<'ctx>,
    builder: &Builder<'ctx>,
    replaced_alloca: &mut HashMap<PointerValue<'ctx>, (PointerValue<'ctx>, IntType<'ctx>)>,
    instr: &InstructionValue<'ctx>,
) -> Result<()> {
    // 1. allocate [ redzone | usable | redzone ]
    let static_arr_kind = instr.get_allocated_type().unwrap();
    if !static_arr_kind.is_array_type() {
        return Ok(());
    }
    let static_arr = static_arr_kind.into_array_type();
    let arr_len = static_arr.len();
    let arr_typ = static_arr.get_element_type();
    let elem_typ = arr_typ.into_int_type();
    let elem_typ_byte = elem_typ.get_bit_width() / 8;
    let rz_size = REDZONE_SIZE / elem_typ_byte;

    builder.position_before(&instr);
    let new_alloca = builder.build_alloca(elem_typ.array_type(rz_size + arr_len + rz_size), "")?;
    let new_alloca_instr = new_alloca.as_instruction().unwrap();
    new_alloca_instr
        .set_alignment(instr.get_alignment().unwrap())
        .unwrap();
    instr.replace_all_uses_with(&new_alloca_instr);
    instr.erase_from_basic_block();

    let next_instr_of_new_alloc = new_alloca_instr.get_next_instruction().unwrap();
    builder.position_before(&next_instr_of_new_alloc);

    // 2. mark redzones and set shadow memory
    build_asan_init_redzone(
        context,
        module,
        builder,
        new_alloca,
```

```
39         static_arr.size_of().unwrap(),
40     )?;
41 
42     // 3. add redzone size to allocated pointer to correctly set the usable pointer
43     let new_alloca_ptr = builder.build_alloca(context.ptr_type(AddressSpace::default()), "")?;
44     let zero_offset = context.i64_type().const_int(0, false);
45     let new_alloca_start_ptr = unsafe {
46         builder.build_in_bounds_gep(
47             elem_typ.array_type(rz_size + arr_len + rz_size),
48             new_alloca,
49             &[zero_offset, zero_offset],
50             "",
51         )?
52     };
53     let rz_offset = context
54         .i64_type()
55         .const_int((REDZONE_SIZE / elem_typ_byte).into(), false);
56     let usable_ptr =
57         unsafe { builder.build_gep(elem_typ, new_alloca_start_ptr, &[rz_offset], "")? };
58     builder.build_store(new_alloca_ptr, usable_ptr)?;
59 
60     replaced_alloca.insert(new_alloca, (new_alloca_ptr, elem_typ));
61     Ok(())
62 }
```

First, we determine the type being allocated. If it's not an array type, there's nothing to instrument, so we return early.

If it is an array, we extract relevant information such as the array's length, its overall type, and the element type. These are needed to construct a new `alloca` instruction that accounts for the additional memory required for redzones.

We then create the new `alloca` instruction with the extended size and replace the original allocation uses using `replace_all_uses_witt()` and `erase_from_basic_block()`, both provided by the LLVM IR binding library (`inkwell` crate).

Once the replacement is complete, we insert a call to the runtime function to initialize the corresponding shadow memory.

Finally, to preserve the original semantics, we adjust the returned pointer so that it points to the start of the usable memory region—just as the original array would. This logic is implemented in lines 42–57.

Since we replaced the original array with the instrumented one, only the `alloca` variable itself is updated. However, arrays are typically accessed through pointers, not directly via the `alloca` instruction—because array indexing requires pointer arithmetic. Therefore, we must also update all pointer-based uses of the original array to reference the instrumented version.

This is handled by the `handle_call()` and `handle_gep()` functions.

> **Remark 4.2** **Summary**
>
> The instrumentation module primarily focuses on three tasks:
> 1. Replacing the original static array with a newly allocated one that includes additional space for redzones.
> 2. Updating all uses of the original array to reference the new, instrumented one.
> 3. Inserting a runtime call to initialize the corresponding shadow memory.
>
> With this, static arrays are fully handled at instrumentation time, while dynamic arrays are managed by the runtime during program execution.

4.2.4.2 ASan Runtime

Runtime tasks are mainly seperated into three:

- **Shadow Memory Initialization**
  - Use special markers to identify memory regions: redzones are marked with a redzone marker, and the usable region is initialized with zeros.
- **Memory Access Assertion**
  - Assert if the memory access is legal
- **Allocator Overloading**
  - Override `malloc()`, `realloc()`, `free()`, and C standard library such as `strcpy()`

Let's take a look how shadow memory is initilized:

```rust
// asan_runtime/src/asan_hook.rs
#[no_mangle]
pub unsafe extern "C" fn __asan_init_redzone(
    raw_ptr: *mut u8,
    usable_size: size_t,
    alloc_kind: u8,
) -> *mut c_void {
    if raw_ptr.is_null() {
        return ptr::null_mut();
    }

    let usable_ptr = unsafe { raw_ptr.add(REDZONE_SIZE) };

    // in a right mannor of implementation, this memset seems redundant
    // without this unused explicit memory initialization, it may trigger segfault
```

```
    // you may try comment out these two lines and use z3 library, then segfault will be triggered
    // when `free` system call is invoked.
    // exact reasoning has not been found yet
    // also, initialization with zero value only works emperically
    libc::memset(raw_ptr as *mut c_void, 0, REDZONE_SIZE);
    libc::memset(raw_ptr.add(REDZONE_SIZE + usable_size) as *mut c_void, 0, REDZONE_SIZE);

    // initialize shadow memory
    poison_shadow_allocated(raw_ptr as usize, usable_size, alloc_kind);
    // record allocated size
    ALLOC_MAP
        .lock()
        .unwrap()
        .insert(usable_ptr as usize, usable_size)
        .expect(ALLOC_MAP_INSERT_ERR_STR);

    // return usable region pointer
    usable_ptr as *mut c_void
}

pub fn convert_to_shadow_idx(addr: usize) -> usize {
    addr >> SHADOW_SCALE
}

fn write_shadow_mem(shadow_mem: &mut [i8], idx: usize, val: i8) {
    shadow_mem[idx % SHADOW_SIZE] = val;
}
```

Redzone memory is initially filled with zeros. The `poison_shadow_allocated()` function is then used to initialize the corresponding shadow memory.

```rust
// asan_runtime/src/asan_hook.rs
lazy_static::lazy_static! {
    pub static ref SHADOW_MEMORY: Mutex<Vec<i8>> = Mutex::new(vec![0i8; SHADOW_SIZE]);
    pub static ref ALLOC_MAP: Mutex<FnvIndexMap<usize, usize, ALLOC_MAP_SIZE>> = Mutex::new(FnvIndexMap::new());
}

/// Poisons left and right redzones and make clean for usable region
```

```
fn poison_shadow_allocated(raw_ptr: usize, usable_size: usize, alloc_kind: u8) {
    // |----------------------|----------------------|------------------
    // ^          LEFT_RZ              USABLE          ^ RIGHT_RZ       ^
    //
    // precisely allocate shadow byte for each boundary(^)

    let usable_ptr = raw_ptr + REDZONE_SIZE;

    let (left_rz_marker, right_rz_marker) = match alloc_kind {
        1 => (STACK_LEFT_REDZONE_MARKER, STACK_RIGHT_REDZONE_MARKER),
        2 => (HEAP_LEFT_REDZONE_MARKER, HEAP_RIGHT_REDZONE_MARKER),
        _ => panic!("unknown alloc kind"),
    };

    // 1. Poison left redzone
    let left_start = raw_ptr;
    let left_end = usable_ptr;
    let shadow_left_start = convert_to_shadow_idx(left_start);
    let shadow_left_end = convert_to_shadow_idx(left_end);
    let mut shadow_mem = SHADOW_MEMORY.lock().unwrap();

    set_bounary_poison_byte(
        &mut shadow_mem,
        left_start,
        shadow_left_start,
        left_rz_marker,
    );
    for i in (shadow_left_start + 1)..shadow_left_end {
        write_shadow_mem(&mut shadow_mem, i, left_rz_marker);
    }

    // 2. Unpoison usable region
    let usable_start = usable_ptr;
    let usable_end = usable_ptr + usable_size;
    let shadow_usable_start = convert_to_shadow_idx(usable_start);
    let shadow_usable_end = convert_to_shadow_idx(usable_end);

    for i in shadow_usable_start..shadow_usable_end {
```

```
            write_shadow_mem(&mut shadow_mem, i, CLEAN_BYTE_MARKER);
        }
        set_bounary_poison_byte(
            &mut shadow_mem,
            usable_end,
            shadow_usable_end,
            right_rz_marker,
        );

        // 3. Poison right redzone
        let right_start = usable_end;
        let right_end = right_start + REDZONE_SIZE;
        let shadow_right_start = convert_to_shadow_idx(right_start);
        let shadow_right_end = convert_to_shadow_idx(right_end);

        for i in (shadow_right_start + 1)..shadow_right_end {
            write_shadow_mem(&mut shadow_mem, i, right_rz_marker);
        }
        set_bounary_poison_byte(
            &mut shadow_mem,
            right_end,
            shadow_right_end,
            right_rz_marker,
        );
    }

    // asan_runtime/src/asan_hook.rs
    fn set_bounary_poison_byte(
        shadow_mem: &mut [i8],
        start: usize,
        shadow_start: usize,
        rz_marker: i8,
    ) {
        let remaining = start & 0x07;
        if remaining != 0 {
            write_shadow_mem(shadow_mem, shadow_start, remaining as i8);
        } else {
            write_shadow_mem(shadow_mem, shadow_start, rz_marker);
        }
    }
```

```
87 // asan_runtime/src/asan_hook.rs
88 pub fn convert_to_shadow_idx(addr: usize) -> usize {
89     addr >> SHADOW_SCALE
90 }
91
92 // asan_runtime/src/asan_hook.rs
93 fn write_shadow_mem(shadow_mem: &mut [i8], idx: usize, val: i8) {
94     shadow_mem[idx % SHADOW_SIZE] = val;
95 }
```

- L17 - 32 fills `LEFT_REDZONE` marker in left redzone area
- L34 - 48 fills `CLEAN_BYTE` marker in usagle region
- L50 - 64 fills `RIGHT_REDZONE` marker in right redzone area

A key point to note is that, as mentioned earlier, precise boundary poisoning is crucial to accurately define the boundaries of each memory region.

`ALLOC_MAP` tracks usable region's pointer and its size where is necessary in `realloc()` and `free()` function.

Up to this point, our ASan runtime has only handled static arrays. Now it's time to extend support to dynamic arrays. Since heap memory is managed through standard C functions like `malloc()`, `realloc()`, and `free()`, we need to intercept and override these calls. The first target for interception is `malloc()`.

```rust
// asan_runtime/src/asan_intrinsic.rs

type MallocFn = unsafe extern "C" fn(size_t) -> *mut c_void;
type ReallocFn = unsafe extern "C" fn(ptr: *mut c_void, size: size_t) -> *mut c_void;
type FreeFn = unsafe extern "C" fn(ptr: *mut c_void);
type StrcpyFn = unsafe extern "C" fn(dest: *mut c_char, src: *const c_char) -> *mut c_char;

lazy_static::lazy_static! {
    static ref ALLOC_MAP: Mutex<HashMap<usize, usize>> = Mutex::new(HashMap::new());
    static ref MALLOC: Mutex<Option<MallocFn>> = Mutex::new(None);
    static ref REALLOC: Mutex<Option<ReallocFn>> = Mutex::new(None);
    static ref FREE: Mutex<Option<FreeFn>> = Mutex::new(None);
    static ref STRCPY: Mutex<Option<StrcpyFn>> = Mutex::new(None);
}

pub fn get_cmalloc() -> MallocFn {
    let mut cmalloc = MALLOC.lock().unwrap();
    if cmalloc.is_none() {
        let buf = get_malloc_cstr();
```

```
        let malloc_sym = unsafe { dlsym(RTLD_NEXT, buf.as_ptr()) };
        let malloc_fn =
            unsafe { std::mem::transmute::<*const (), MallocFn>(malloc_sym as *const ()) };
        *cmalloc = Some(malloc_fn);
    }
    cmalloc.unwrap()
}

// asan_runtime/src/asan_hook.rs

thread_local! {
    pub static MALLOC_REENTERED: Mutex<bool> = const { Mutex::new(false) }
}

#[no_mangle]
pub unsafe extern "C" fn malloc(size: size_t) -> *mut c_void {
    let cmalloc = get_cmalloc();

    MALLOC_REENTERED.with(|re_enter| {
        let is_renter = re_enter.lock().unwrap();
        if *is_renter {
            cmalloc(size)
        } else {
            let total_size = size + REDZONE_SIZE * 2;
            let raw_ptr = cmalloc(total_size) as *mut u8;
            __asan_init_redzone(raw_ptr, size,  ALLOC_HEAP)
        }
    })
}
```

The main responsibility of the overridden `malloc()` function is to allocate additional memory to account for both left and right redzones, and then initialize the corresponding shadow memory.

The standard memory allocation process still applies. Therefore, we must request a larger allocation from the operating system. To do this, the first step is to retrieve the original `malloc()` function dynamically from the symbol table.

Once the original `malloc()` is obtained, we request a total allocation of `size + 2 * REDZONE_SIZE`, and then call the shadow memory initialization routine with the allocation kind `ALLOC_HEAP` which indicates whether the memory is stack or heap.

For other standard functions, the same process is repeated per function.

```rust
// asan_runtime/src/asan_intrinsic.rs
```

```

pub fn get_malloc_cstr() -> [i8; 7] {
    let fn_str = b"malloc";
    let mut buf = [0 as c_char; 7];
    for (i, &b) in fn_str.iter().enumerate() {
        buf[i] = b as c_char;
    }
    buf
}

pub fn get_realloc_str() -> [i8; 8] {
    let fn_str = b"realloc";
    let mut buf = [0 as c_char; 8];
    for (i, &b) in fn_str.iter().enumerate() {
        buf[i] = b as c_char;
    }
    buf
}

pub fn get_free_str() -> [i8; 5] {
    let fn_str = b"free";
    let mut buf = [0 as c_char; 5];
    for (i, &b) in fn_str.iter().enumerate() {
        buf[i] = b as c_char;
    }
    buf
}

pub fn get_strcpy_str() -> [i8; 7] {
    let fn_str = b"strcpy";
    let mut buf = [0 as c_char; 7];
    for (i, &b) in fn_str.iter().enumerate() {
        buf[i] = b as c_char;
    }
    buf
}

pub fn get_cmalloc() -> MallocFn {
    let mut cmalloc = MALLOC.lock().unwrap();
    if cmalloc.is_none() {
        let buf = get_malloc_cstr();
```

```
        let malloc_sym = unsafe { dlsym(RTLD_NEXT, buf.as_ptr()) };
        let malloc_fn =
            unsafe { std::mem::transmute::<*const (),
            MallocFn>(malloc_sym as *const ()) };
        *cmalloc = Some(malloc_fn);
    }
    cmalloc.unwrap()
}

pub fn get_crealloc() -> ReallocFn {
    let mut crealloc = REALLOC.lock().unwrap();
    if crealloc.is_none() {
        let buf = get_realloc_str();
        let realloc_sym = unsafe { dlsym(RTLD_NEXT, buf.as_ptr()) };
        let realloc_fn =
            unsafe { std::mem::transmute::<*const (),
            ReallocFn>(realloc_sym as *const ()) };
        *crealloc = Some(realloc_fn);
    }
    crealloc.unwrap()
}

pub fn get_cfree() -> FreeFn {
    let mut cfree = FREE.lock().unwrap();
    if cfree.is_none() {
        let buf = get_free_str();
        let free_sym = unsafe { dlsym(RTLD_NEXT, buf.as_ptr()) };
        let free_fn = unsafe { std::mem::transmute::<*const (),
        FreeFn>(free_sym as *const ()) };
        *cfree = Some(free_fn);
    }
    cfree.unwrap()
}

pub fn get_strcpy() -> StrcpyFn {
    let mut cstrcpy = STRCPY.lock().unwrap();
    if cstrcpy.is_none() {
        let buf = get_strcpy_str();
        let strcpy_sym = unsafe { dlsym(RTLD_NEXT, buf.as_ptr()) };
        let strcpy_fn =
            unsafe { std::mem::transmute::<*const (),
            StrcpyFn>(strcpy_sym as *const ()) };
```

```rust
            *cstrcpy = Some(strcpy_fn);
        }
        cstrcpy.unwrap()
    }
```

`free()` implementation is as follows:

```rust
// asan_runtime/src/asan_hook.rs
#[no_mangle]
pub unsafe extern "C" fn free(ptr: *mut c_void) {
    if ptr.is_null() {
        return;
    }
    let cfree = get_cfree();
    let mut alloc_map = ALLOC_MAP.lock().unwrap();
    if let Some(size) = alloc_map.remove(&(ptr as usize)) {
        poison_shadow_freed(ptr, size);
        cfree(ptr.sub(REDZONE_SIZE));
    } else {
        cfree(ptr);
    }
}
```

The first step is to obtain the standard `free()` symbol. Since our custom `malloc()` handler records each allocation in the `ALLOC_MAP` structure—including the usable pointer and its size—we can track all memory managed by our runtime.

When `free()` is called, we check whether the pointer was allocated through our `malloc()` handler by querying `ALLOC_MAP`. If it was, we retrieve the allocation size and mark the entire region—including the usable portion and both redzones—with a `FREED` marker.

Any subsequent access to this freed region will be detected by our access checker, allowing us to report UAF. The process of marking shadow memory as `FREED` is as follows:

```rust
// asan_runtime/src/asan_hook.rs
/// Make all regions as poisoned
fn poison_shadow_freed(usable_ptr: *mut c_void, size: usize) {
    let start = unsafe { usable_ptr.sub(REDZONE_SIZE) };
    let shadow_start = convert_to_shadow_idx(start as usize);
    let shadow_end =
        unsafe { convert_to_shadow_idx(start.add(REDZONE_SIZE + size + REDZONE_SIZE) as usize) };
    let mut shadow = SHADOW_MEMORY.lock().unwrap();
    set_bounary_poison_byte(&mut shadow, start as usize, shadow_start, FREED_MARKER);
```

```
    for i in (shadow_start + 1)..shadow_end {
        write_shadow_mem(&mut shadow, i, FREED_MARKER);
    }
    set_bounary_poison_byte(&mut shadow, start as usize, shadow_end, FREED_MARKER);
}
```

The third function we intercept is `realloc()`. The behavior of `realloc()` can be broken down into two main tasks:

1. Allocate a new memory block of the requested size and copy the original data into it.
2. Free the original memory block.

Our implementation must preserve this behavior while adding shadow memory handling. Specifically, we invoke our custom `malloc()` to perform the new allocation and initialize the corresponding shadow memory. Then, we call our custom `free()` handler to mark the original memory (including its redzones) as `FREED` in shadow memory.

```rust
// asan_runtime/src/asan_hook.rs
#[no_mangle]
pub unsafe extern "C" fn realloc(ptr: *mut c_void, size: size_t) -> *mut c_void {
    if ptr.is_null() {
        return malloc(size);
    }
    let mut alloc_map = ALLOC_MAP.lock().unwrap();
    if let Some(org_size) = alloc_map.remove(&(ptr as usize)) {
        // drop locked variables because the following code will require `free` call
        drop(alloc_map);
        poison_shadow_freed(ptr, size);
        let usable_ptr = malloc(size);
        ptr::copy(ptr, usable_ptr, org_size);
        free(ptr.sub(REDZONE_SIZE));
        usable_ptr
    } else {
        let crealloc = get_crealloc();
        crealloc(ptr, size)
    }
}
```

Now, our implementation til now handles both static and dynamic array.

```c
#include <stdlib.h>

int main() {
```

```c
4      int* arr = (int*)malloc(sizeof(int) * 3);
5      arr[0] = 1;
6      arr[1] = 1;
7      arr[2] = 1;
8      arr[3] = 1; // OOB report
9  }
```

OOB report at line number 8:

```
❯ LD_LIBRARY_PATH=./ ./a.out
[ASAN] invalid memory access detected at kkk.c: 0x640c601292cc
   0: asan_runtime::asan_runtime::report_asan_violated::{{closure}}
             at .../asan_runtime/src/asan_runtime.rs:46:18
   1: std::thread::local::LocalKey<T>::try_with
             at /build/rustc-1.84.1-src/library/std/src/thread/local.rs:283:12
   2: std::thread::local::LocalKey<T>::with
             at /build/rustc-1.84.1-src/library/std/src/thread/local.rs:260:9
   3: asan_runtime::asan_runtime::report_asan_violated
             at .../asan_runtime/src/asan_runtime.rs:44:5
   4: __asan_mem_check
             at .../asan_runtime/src/asan_runtime.rs:30:9
   5: main
             at ./asan-example2.c:8:12
   6: __libc_start_call_main
   7: __libc_start_main_alias_2
   8: _start
```

Now the memory range assertion is introduced:

```rust
// asan_runtime/src/asan_runtime.rs
#[no_mangle]
pub extern "C" fn __asan_mem_check(file_ptr: *const libc::c_char, addr: usize, access_size: usize) {
    if file_ptr.is_null() || addr == 0 {
        return;
    }

    let filename = cstr_to_string(file_ptr);
    let shadow_idx = convert_to_shadow_idx(addr);
    let shadow_val = SHADOW_MEMORY.lock().unwrap()[shadow_idx % SHADOW_SIZE];
    if shadow_val != CLEAN_BYTE_MARKER && ((addr & 0x07) + access_size) as i8 > shadow_val {
```

```
        report_asan_violated(&filename, addr);
    }
}

fn report_asan_violated(filename: &str, addr: usize) {
    if is_test_enabled() {
        eprintln!("[ASAN] invalid memory access detected at {}", filename);
    } else {
        eprintln!(
            "[ASAN] invalid memory access detected at {}: 0x{:x}",
            filename, addr
        );
    }
    // print backtrace
    MALLOC_REENTERED.with(|re_enter| {
        *re_enter.lock().unwrap() = true;
        let bt = Backtrace::force_capture();
        if is_test_enabled() {
            eprintln!("{}", trim_runtime_bt(bt.to_string()));
        } else {
            eprintln!("{bt}");
        }
        *re_enter.lock().unwrap() = false;
    });
    if !is_test_enabled() {
        unsafe {
            libc::_exit(EXIT_CODE);
        }
    }
}

fn is_test_enabled() -> bool {
    matches!(env::var(ASAN_TEST_ENABLED), Ok(val) if val == "1")
}

fn trim_runtime_bt(bt: String) -> String {
    bt.lines()
        .skip_while(|line| !line.contains("__asan_mem_check"))
        .skip(2)
        .collect::<Vec<_>>()
```

```
        .join("\n")
}
```

An important detail to note is the use of `MALLOC_REENTERED`. The function `poison_shadow_allocated()` may itself trigger memory allocation when initializing shadow memory. Although shadow memory is declared globally, it is not fully allocated up front—memory is committed on-demand, page by page.

When an OOB violation is reported, functions like `eprintln!` may also perform dynamic memory allocation. This leads to a call to our overridden `malloc()`, which can result in a recursive loop and ultimately cause a stack overflow.

To prevent this, we use `MALLOC_REENTERED` to detect and break the recursion. During reporting, we bypass the normal path and allow a minimal, direct call of standard `malloc()`.

This does not compromise shadow memory management, as the allocation is unrelated to the target program's memory space and occurs just before the program terminates due to the reported error.

Lastly, an important point to note: since each redzone is 32 bytes, totaling 64 bytes, any access beyond the redzone boundaries will go undetected by the runtime. In other words, if OOB access occurs outside the redzones, no report will be triggered.

You can test this behavior by modifying line 8 as follows:

```c
    arr[50] = 1; // Not reported
```

However, there are certain edge cases that our implementation cannot currently detect. For example, consider the following scenario:

```c
#include <string.h>

int main() {
    char buffer[10];
    strcpy(buffer, "aaaaaaaaaa");
    return 0;
}
```

This is clearly an OOB bug. However, the standard `strcpy()` function is part of the C runtime and executes outside our instrumented code, making it invisible to our current instrumentation. Similar issues can arise when other standard library functions perform memory operations outside the scope of our target program code.

To address this, we can intercept these C runtime functions—just as we did with `malloc()`, `free()`, and `realloc()`—by overriding them. The following is our custom implementation of `strcpy()`.

```rust
#[no_mangle]
pub unsafe extern "C" fn strcpy(dest: *mut c_char, src: *const c_char) -> *mut c_char {
    let cstrcpy = get_strcpy();
    let src_len = CStr::from_ptr(src).to_bytes().len();
    let temp_filename_ptr = c"libc::strcpy".as_ptr() as *const c_char;
```

```
6       for i in 0..=src_len {
7           __asan_mem_check(temp_filename_ptr, dest as usize + i, 1);
8       }
9       cstrcpy(dest, src)
10  }
```

The core behavior of `strcpy()` involves copying elements one by one by iterating through the source array. We insert memory access checks for each index of the destination buffer (as shown in lines 6–8). After verifying the safety of all accesses, we delegate the actual copying to the original `strcpy()` function provided by the C runtime.

4.2.4.3 Summary

We implemented OOB detector using both compile-time instrumentation and a runtime library. The instrumentation module inserts runtime function calls that assert the validity of memory accesses, while the runtime intercepts memory allocation functions and manages shadow memory.

You can refer to the full implementation here:

- Instrumentation module: `instrument/src`
- Runtime library: `asan_runtime/src`

Test cases are available in the `coverage/tools/tests/inputs/asan directory`

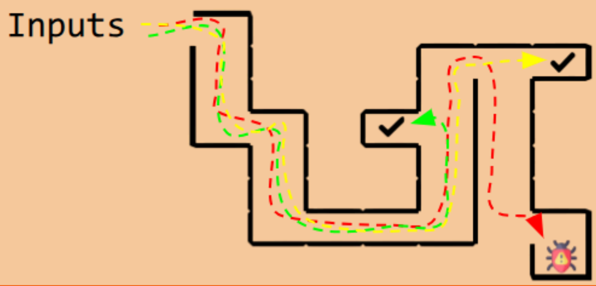
4.2.4.4 Homework

Exercise 4.1 We suggest the following topics for further study and exploration:

- **Topic #1: Global object handling**
  - Our implementation supports both static and heap-allocated arrays. For static arrays, we currently handle only local static arrays, not global ones. However, support for global arrays can be added by extending the instrumentation at the frontend level. We encourage readers to explore this as a possible enhancement.
- **Topic #2: C standard library support**
  - This book demonstrates how to override the C standard library function `strcpy()` as an example. Readers are encouraged to extend this approach to other functions such as `strlen()`, `strcmp()`, and beyond.

Part IV
Fuzzing

“Automating the process of finding a program’s vulnerabilities”



5. Fuzzer

5.1 Introduction to Fuzzer

Fuzzing [21] is one of the most effective software testing methods and a well-known approach for discovering bugs. It is widely used in both academia and industry.

A naive fuzzer works by generating randomized inputs and feeding them to the target program until a crash is detected.

Let’s imagine what fuzzing would be. The example code below takes user input via `stdin` and evaluate the value. If the value is 10, then prgoram will crash, otherwise, print the value to terminal. Assume that we don’t know the crash condition (i.e, when `input()` evaluates to `10`)

```
v = input()
if (v == 10) {
  crash()
} else {
  printf("%d", v);
}
```

An example program waits for input. A fuzzer generates random input and feeds it to the program through `stdin`. The program, previously blocked waiting for input, then proceeds to execute the next portion of its code. Suppose the input is `1`; in this case, the program exits normally, indicating no crash. The fuzzer then moves to the next round—say with input 2—and continues generating and feeding inputs. This process is repeated until a crash is triggered.

This represents the original idea behind fuzzing when it was first introduced. Since then, fuzzers have evolved significantly to find crashes more effectively. The figure below illustrates the basic concept of a naive fuzzer versus a modern, off-the-shelf fuzzer.

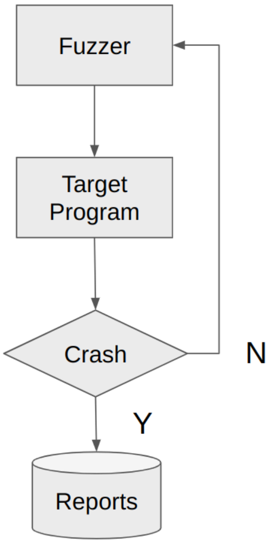


Figure 5.1: A naive fuzzer

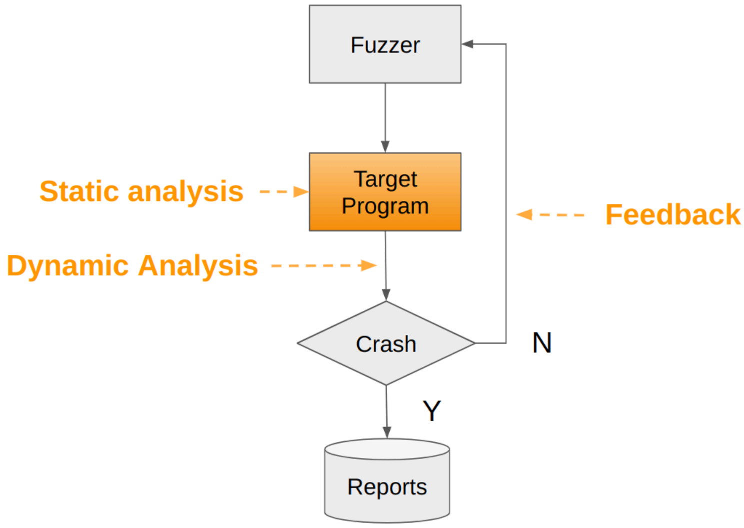


Figure 5.2: Advanced fuzzer

The left figure illustrates an imaginary naive fuzzer, while the right figure shows an off-the-shelf fuzzer that leverages both static and dynamic analysis to understand program structure. These advanced techniques help guide the fuzzing process and significantly improve the effectiveness of the fuzzing campaign. Modern fuzzers contribute not only in generating inputs but also in analyzing programs to extract useful information—such as probability-based heuristics, algorithmic insights, or theoretical models—to make fuzzing more intelligent and targeted.

Fuzzers can be broadly categorized into three types: black-box, white-box, and gray-box.

- A **black-box fuzzer**, as the name suggests, treats the program as an opaque system. It only observes output behavior in response to given inputs.
- A **white-box fuzzer** theoretically has access to all internal details of the program. It can leverage static and dynamic analysis to understand and exploit the program's behavior for optimal input generation.
- A **gray-box fuzzer** operates in between, utilizing partial information—most commonly, code coverage—to guide input generation.

In this book, we will implement a **gray-box fuzzer**.

There are several factors that contribute to an effective fuzzer, with coverage being one of the most widely used and important metrics.

American Fuzzy Lop (AFL) [22] is one of the most popular coverage-based fuzzers. It instruments the target program to collect coverage information at runtime, which the fuzzer then uses to guide its exploration toward more interesting code paths.

It's important to note that *coverage-based* does not imply that the fuzzer actively increases code coverage. Instead, **it tracks which parts of the code have been executed and identifies whether new code paths have been discovered. Inputs that reach previously unseen or rarely executed code are assigned higher scores.** In essence, a coverage-based fuzzer determines its fuzzing strategy based on code coverage feedback.

In this book, we use AFL as our reference model and implement an AFL-like fuzzer in Rust. As in previous chapters, our fuzzer consists of two main components: instrumentation and a runtime library. Let's begin by exploring the design of our fuzzer.

5.2 A Big Picture

The figure below illustrates the fuzzer we will implement throughout this book. Symbolic execution and crash minimization will be introduced in Chapters 4 and 5, respectively.

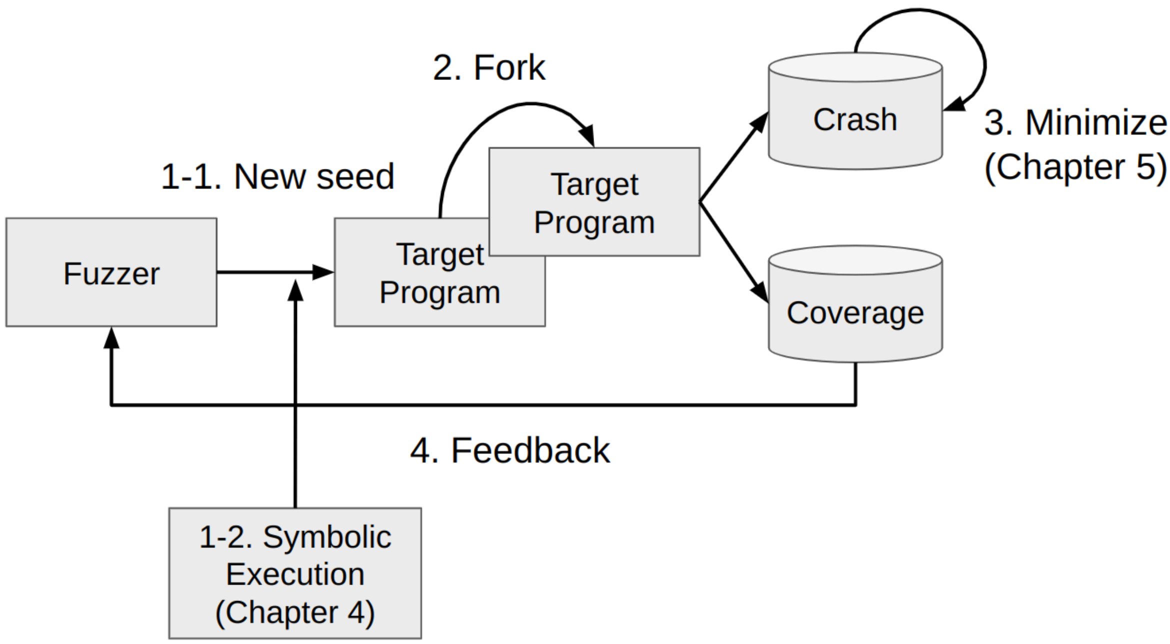


Figure 5.3: A workflow of fuzzing campaign

The fuzzer is an independent binary that operates separately from the instrumentation, runtime, and target program. It generates a new seed input, which is sent to the target program. The target program remains idle until it receives this input from the fuzzer. Upon receiving the seed, it forks itself, and the child process executes using the provided input.

After the runtime is invoked, the execution can result in two outcomes: a crash or coverage information. If the target program crashes, the input is considered failure-inducing. Before saving this input to the file system, a minimizer attempts to reduce it to the smallest form that still triggers the crash, making it easier for developers to analyze the root cause.

The second type of output is the coverage footprint, which indicates which parts of the code were newly explored, including the specific edges visited. The fuzzer uses this information to guide the next iteration by generating a new seed that aims to expand code coverage.

This sequence of generating input, executing the target, collecting output, and guiding further mutations constitutes a single fuzzing iteration, which is then repeated continuously. In addition, the target program's input can also be generated by a symbolic execution runtime, which will be introduced in Chapter 4.

5.3 Design and Architecture

5.3.1 Branch(Edge) Coverage

The first important topic is branch (or edge) coverage. As explained in this LLVM-IR tutorial, all instructions within a basic block in LLVM IR are guaranteed not to alter control flow. In AFL, the basic block serves as the fundamental unit of coverage. Consider the following example:

```llvm
bb1:
  ...
  br i1 %x, label %bb2, label %bb3
bb2:
  ...
bb3:
  ...
```

There are three basic blocks, and the possible coverage paths are either `bb1 -> bb2` or `bb1 -> bb3`. Coverage is recorded by instrumenting each basic block using the instrumentation module. This coverage data is stored in shared memory, which is accessible to both the fuzzer and its runtime. Each basic block is instrumented as follows:

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

The cur_location is a randomly generated value assigned at compile time. `shared_mem` refers to the shared memory region used by both the fuzzer and the runtime to track coverage. The index of the current edge is calculated using the XOR operation: `cur_location ^ prev_location`. This XORed value uniquely identifies the edge in the control flow. For example, the edges `bb1 -> bb3` and `bb2 -> bb3` produce different XOR values and are therefore treated as distinct. Let's deep dive into the semantic.

Basic block	Current location	Prev location	Shared memory index
A	**r1**	0	r1 ^ 0 = r1
B	**r2**	r1 >> 1	r2 ^ (r1 >> 1)
C	**r3**	r2 >> 1	r3 ^ (r2 >> 1)

The table assumes the execution flow `A -> B -> C`. Each basic block is first assigned a unique random identifier at compile time. To track which path (or edge) has been executed, the runtime calculates `cur_location ^ prev_location`, effectively representing the transition from one basic block to another as a unique tuple. For example, in the path `A → B`, only one such tuple (`AB`) is recorded. A second execution of `A -> B` will compute the same XOR value and thus update the same index in the shared memory.

```
`A -> B -> A -> B`
```

First, `A -> B` creates a new index of the shared memory. `B -> A` also hits a new index. The remaining one is only `A -> B` again. However, at this moment, the calculation creates an index before we seen when we create a first `A -> B`. Then it increases the count of tuple `AB`. Thus, the shared memory is imagine as follows:

First, the transition `A -> B` generates a new index in the shared memory. Later, `B -> A` results in a different index, representing a distinct edge. When `A -> B` occurs again, the XOR calculation produces the same index as the initial `A -> B` transition. Instead of creating a new entry, it increments the counter for the existing `AB` tuple. As a result, the shared memory can be visualized as follows:

```
shared_mem[index1] = 2 // A -> B
shared_mem[index2] = 1 // B -> A
```

Thus, the second `A -> B` transition does not represent new coverage—it is not a new discovery. Finally, let's look at another example provided in the AFL documentation.

```
#1: A -> B -> C -> D -> E
#2: A -> B -> C -> A -> E
```

Assume that two paths have already been discovered, and the current path does not introduce any new coverage.

```
#3: A -> B -> C -> A -> B -> C -> A -> B -> C -> D -> E
```

Let's break down the path into tuples, one step at a time:
- A -> B (#1)
- B -> C (#1)
- C -> D (#1)
- D -> E (#1)
- C -> A (#2)
- A -> E (#2)

We can easily identify that step #3 does not represent new coverage.

5.3.2 Bucketing

As explained earlier, coverage information determines the direction of fuzzing. Coverage-based fuzzers primarily prioritize two things: (1) discovering new paths and (2) exploring less frequently executed paths.

A new path is valuable because it represents code that has never been executed before, making it a prime candidate for uncovering potential bugs. For the second case, consider the probability of finding a bug: if you have executed code path A many times but rarely or never executed path B, it makes sense to focus on path B since it is less explored and may hide undiscovered issues.

In summary, discovering new paths is the highest priority. If no new paths are found, the fuzzer then prioritizes paths that have been executed less frequently.

To represent how many times a path is executed, a coarse-grained bucketing scheme is introduced. The bucket consists of `8` elements representing execution count ranges: `1, 2, 3, 4-7, 8-15, 16-31, 32-127, and 128+`. Each bucket index is assigned a score from `8` (for the lowest count) down to `1` (for the highest count). For example, if a path is executed only once, it receives a score of `8`; if it is executed more than 128 times, it receives a score of `1`.

When new coverage is discovered, it is awarded a score higher than 8 to prioritize novel paths.

This process of grouping execution counts into discrete ranges and assigning scores is called bucketing.

5.3.3 Fork Server

When the fuzzer generates a new seed, it is fed to the target program. However, if the target program is reloaded as a new process for each seed, a significant amount of time is wasted on process startup overhead. In general, the following tasks are involved each time a new process is loaded:

- Disk I/O: Read the binary from disk and loads into memory
- Initialization:
  - Global variable / Heap / Stack
  - Linker / Loader
  - Resource configuration (file descriptor / socket / etc)

To avoid the repeated overhead of loading a new process for every seed, the fork-server mechanism is introduced.

There are various ways to implement a fork-server. In this book, it is implemented within the runtime. The runtime initially loads the target program into memory once and never reloads it afterward. After loading, the process remains blocked, waiting for a signal—let's call it *wakeup*. Upon receiving this signal, the process forks itself, resulting in two processes: a parent and a child.

The parent process remains blocked until the child process finishes execution. Meanwhile, the child process runs the main logic of the target program using the provided seed. Once the child exits, the parent collects the exit status and signals it back to the fuzzer. This cycle repeats for each seed.

By doing this, the fork-server eliminates the costly process initialization step. Instead of starting a fresh process each time, it efficiently forks from a pre-initialized state. Communication between the fuzzer and the fork-server can be handled via a dedicated shared memory region, which will be explained in detail in the implementation section.

5.3.4 Harness

Think about any program you want to fuzz—and how to feed fuzzing input into it.

For a target program to accept input from an external source like a fuzzer, it must expose an interface for receiving that input. However, most programs are not designed to cooperate with fuzzing by default. Therefore, we often need to modify or wrap the original target program to enable fuzzing interaction. This wrapper or interface is called a ***harness***.

A harness must be tailored to each specific program. For example, a GUI program may require simulating user interactions such as mouse clicks or keystrokes, while an IoT program might need to simulate external sensor data.

In general, for text user interface (TUI) programs, two common methods are used to deliver input: via `stdin` or via files.

For `stdin`, the harness directs the fuzzing input to the program's standard input file descriptor. The program reads from `stdin` and processes the input through its main logic.

For file-based input, the harness writes the fuzzing input to a file, and the target program reads from that file as if it were regular input. The file content is updated for each fuzzing iteration.

In this book, we focus on two types of harnesses: `stdin`-based and file-based.

5.3.5 Oracle

If a program crashes, how do we determine whether it completed normally or actually crashed?

To answer this, we need a clear definition of what constitutes a crash. In many security papers, researchers define an oracle—a mechanism for identifying abnormal or interesting behaviors in the target program.

A common oracle is a segmentation fault. If a segmentation fault occurs, we can confidently say the program crashed. However, the absence of a segmentation fault does not necessarily mean the program executed correctly. For instance, in Chapter 2, we discussed OOB bugs: one of the bug category such as segmentation fault. Unfortunately, in C programs, certain OOB accesses may not immediately trigger visible crashes or errors.

To detect such hidden bugs, the target program can be compiled with sanitizers like ASan. With ASan enabled, the program will crash explicitly when an OOB access occurs. The fuzzer can then detect this by checking the program's exit status.

The key principle is this: when you define an oracle, the target program must be instrumented or configured to detect and report the kind of illegal behavior the oracle is designed to capture.

5.3.6 Fuzzing Campaign

We have introduced the core components of the fuzzing process. Now, it's time to put all the pieces together and illustrate the complete workflow.

1. The fuzzer initializes shared memory for communication between itself and the runtime.
2. The fuzzer loads the target process (which remains blocked, waiting for input).
3. The fuzzer feeds an initial seed to the target by waking up the fork-server.
4. The target program forks, and the child process executes the main logic using the provided seed.
5. The fuzzer collects the child process's exit status and coverage information, then evaluates a score for the given seed.
6. The fuzzer checks whether the seed caused a crash. If so, it minimizes the input (as described in the next chapter) and saves it to disk.
7. The fuzzer generates a new seed and returns to step 1.

This cycle is often referred to as a fuzzing campaign.

Topics such as crash minimization and symbolic execution will be explored in detail in the following chapters.

5.4 Implementation

As with previous components, two parts are required: instrumentation and runtime. However, at this stage, we also need to develop an additional, standalone binary—the fuzzer itself. Let's start by exploring the instrumentation component to record basic branch coverage.

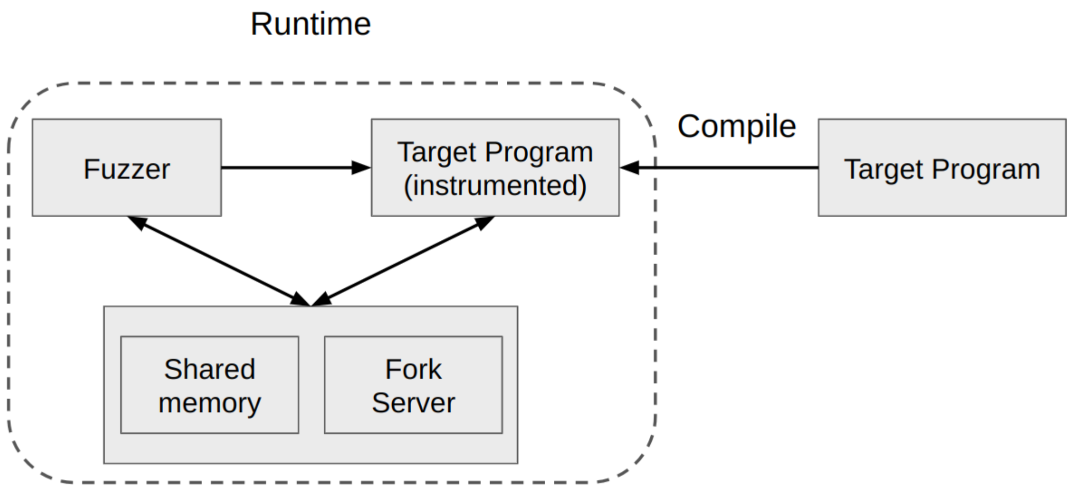


Figure 5.4: Implementation Components

5.4.1 Instrumentation

The instrumentation module is lightweight and primarily handles two tasks: initializing the fork-server and measuring code coverage.

5.4.1.1 Fork-server Initialization

Fork-server initialization sets up the target program to receive signals from the fuzzer. Once initialized, it simply blocks and waits until a signal is received. That's all it does.

This initialization routine in the runtime is triggered via a constructor function.

```rust
// instrument/src/fuzzer.rs
pub fn build_fuzzer_init<'ctx>(
    context: &'ctx Context,
    module: &Module<'ctx>,
    builder: &Builder<'ctx>,
) -> Result<FunctionValue<'ctx>> {
    // __fuzzer_forkserver_init() - initialize fuzzer
    let init_func_typ = context.void_type().fn_type(&[], false);
    let init_fn = module.add_function(FUZZER_FORKSERVER_INIT,
    init_func_typ, None);

    // Create a constructor function to initialize fuzzer
    let constructor = module.add_function(FUZZER_MODULE_INIT,
    init_func_typ, None);
    let entry = context.append_basic_block(constructor,
    FUZZER_INIT_ENTRY);
    builder.position_at_end(entry);
    builder.build_call(init_fn, &[], "")?;
    builder.build_return(None)?;
```

```
    Ok(constructor)
}
```

It defines and creates the function `__fuzzer_module_init()`, which consists of a single basic block that calls the runtime function `__fuzzer_forkserver_init()`. The following LLVM IR snippet is generated by the `build_fuzzer_init()` function.

LLVM-IR
```llvm
declare void @__fuzzer_forkserver_init()

define void @__fuzzer_module_init() {
__fuzzer_init_entry:
  call void @__fuzzer_forkserver_init()
  ret void
}
```

5.4.1.2 Basic Block(Edge) Coverage

The remaining task is to instrument the runtime function to record coverage for each basic block.

Rust
```rust
// instrument/src/fuzzer.rs
fn get_rand_value<'ctx>(context: &'ctx Context) -> IntValue<'ctx> {
    let (lower, _) = Uuid::new_v4().as_u64_pair();
    let i64_typ = context.i64_type();
    i64_typ.const_int(lower, false)
}

#[derive(Default)]
pub struct FuzzModule {}

impl InstrumentModule for FuzzModule {
    fn instrument<'ctx>(
        &self,
        context: &'ctx Context,
        module: &Module<'ctx>,
        builder: &Builder<'ctx>,
    ) -> Result<()> {
        let constructor = build_fuzzer_init(context, module, builder)?;
        build_ctros(context, module, constructor)?;

        let module_filename = cstr_to_str(module.get_source_file_name());
        let funcs: Vec<_> = module.get_functions().collect();
        for func in funcs {
```

```rust
            // Skip funcs without bodies or those we've added
            if can_skip_instrument(&func) {
                continue;
            }
            let mut instrumented_blks = HashSet::new();
            for basic_blk in func.get_basic_blocks() {
                if instrumented_blks.contains(&basic_blk) {
                    continue;
                }
                if let Some(first_instr) = basic_blk.get_first_instruction() {
                    let mut instrument_pos = first_instr;
                    match first_instr.get_opcode() {
                        InstructionOpcode::LandingPad | InstructionOpcode::Phi => {
                            if let Some(second_instr) = first_instr.get_next_instruction() {
                                instrument_pos = second_instr;
                            }
                        }
                        _ => {}
                    }
                    if let Some(instr_filename) = get_instr_filename(&instrument_pos) {
                        if instr_filename != module_filename {
                            continue;
                        }
                    }
                    builder.position_before(&instrument_pos);
                    build_trace_edge(context, module, builder, get_rand_value(context))?;
                }
                instrumented_blks.insert(basic_blk);
            }
        }
        // Verify instrumented IRs
        module_verify(module)
    }
}

// instrument/src/inkwell_intrinsic.rs
fn get_trace_edge<'ctx>(context: &'ctx Context, module: &Module<'ctx>) -> FunctionValue<'ctx> {
```

```
61      match get_func(module, FUZZER_TRACE_EDGE) {
62          Some(func) => func,
63          None => {
64              let trace_edge = context
65                  .void_type()
66                  .fn_type(&[context.i64_type().into()], false);
67              module.add_function(FUZZER_TRACE_EDGE, trace_edge, None)
68          }
69      }
70  }
71
72  pub fn build_fuzzer_init<'ctx>(
73      context: &'ctx Context,
74      module: &Module<'ctx>,
75      builder: &Builder<'ctx>,
76  ) -> Result<FunctionValue<'ctx>> {
77      // __fuzzer_forkserver_init() - initialize fuzzer
78      let init_func_typ = context.void_type().fn_type(&[], false);
79      let init_fn = module.add_function(FUZZER_FORKSERVER_INIT, init_func_typ, None);
80
81      // Create a constructor function to initialize fuzzer
82      let constructor = module.add_function(FUZZER_MODULE_INIT, init_func_typ, None);
83      let entry = context.append_basic_block(constructor, FUZZER_INIT_ENTRY);
84      builder.position_at_end(entry);
85      builder.build_call(init_fn, &[], "")?;
86      builder.build_return(None)?;
87      Ok(constructor)
88  }
```

The implementation is straightforward. It inserts a call to the `__fuzzer_trace_edg()` function at lines 48–49, which is handled by the runtime.

The resulting instrumented LLVM IR looks as follows:

```
LLVM-IR
bb1:
  ...
  call void @__fuzzer_trace_edge(i64 -2510630191682466935)
  ...

bb2:
  ...
```

```
  call void @__fuzzer_trace_edge(i64 6646994726620055515)
  ...

...
```

The argument passed to `__fuzzer_trace_edge()` corresponds to `cur_location = <COMPILE_TIME_RANDOM>`. This means that for each basic block, a compile-time generated random value is assigned and used as the current location during runtime. With the instrumentation complete, we now move on to implementing the runtime library.

5.4.2 Runtime

There are only two external functions.

```rust
// fuzzer_runtime/src/runtime.rs
#[no_mangle]
pub extern "C" fn __fuzzer_trace_edge(cur_loc: i64) {
    EDGE_COVERAGE.lock().unwrap().trace_edge(cur_loc);
}

#[no_mangle]
pub extern "C" fn __fuzzer_forkserver_init() {
    EDGE_COVERAGE.lock().unwrap().read_wakeup();
}
```

Before looking into the body of `read_wakeup()`, let's take a look the data structure.

```rust
// fuzzer_runtime/src/coverage.rs
lazy_static::lazy_static! {
    pub static ref EDGE_COVERAGE: Arc<Mutex<EdgeCoverage>> = Arc::new(Mutex::new(EdgeCoverage::new()));
}

pub struct EdgeCoverage {
    shm: Option<MmapMut>,
    shm_size: Option<usize>,
    aux: Option<MmapMut>,
    fork_server_host: Option<i32>,
    fork_server_runtime: Option<i32>,
}

impl Default for EdgeCoverage {
    fn default() -> Self {
        Self::new()
    }
```

```
}

impl EdgeCoverage {
    pub fn new() -> Self {
        let (shm_mmap, shm_size) = init_shm("SHM_ID", "SHM_SIZE");
        let (shm_aux_mmap, _) = init_shm("SHM_AUX_ID",
        "SHM_AUX_SIZE");
        Self {
            shm: shm_mmap,
            shm_size,
            aux: shm_aux_mmap,
            fork_server_host:
            init_forkserver_fd("FORK_SERVER_HOST"),
            fork_server_runtime:
            init_forkserver_fd("FORK_SERVER_RUNTIME"),
        }
    }

    /// only used for test
    pub fn init(&mut self) {
        let new_edge_cov = Self::new();
        self.shm = new_edge_cov.shm;
        self.shm_size = new_edge_cov.shm_size;
        self.aux = new_edge_cov.aux;
        self.fork_server_host = new_edge_cov.fork_server_host;
        self.fork_server_runtime = new_edge_cov.fork_server_runtime;
    }

    ...
}
```

`EdgeCoverage` is implemented as a singleton object. The `shm` and `aux` variables are file descriptors pointing to shared memory regions. These shared memory segments are mapped to the `/tmp` directory using `mmap()` [23]. The actual creation and mapping of this virtual memory space are delegated to the fuzzer.

The `shm` region stores execution data—specifically, which basic blocks have been executed and how many times. The `aux` region holds auxiliary metadata, such as the total number of edges, which edges are discovered and what's are new finding.

The `fork_server_host` and `fork_server_runtime` are file descriptors used for event notification between the fuzzer and the runtime. Signaling between the two is performed via these event file descriptors, using mechanisms such as eventfd [24].

```rust
// fuzz_runtime/src/internal.rs
pub fn set_mmap(id: &str) -> Option<MmapMut> {
```

```rust
    match env::var(id) {
        Ok(path) => {
            let file = OpenOptions::new()
                .read(true)
                .write(true)
                .open(path)
                .expect("Failed to open SHM file");
            let mmap = unsafe { MmapMut::map_mut(&file).expect("mmap failed in runtime") };
            Some(mmap)
        }
        Err(_) => None,
    }
}

pub fn init_shm(id: &str, size_id: &str) -> (Option<MmapMut>, Option<usize>) {
    let ret = set_mmap(id);
    let mut shm_size = None;
    if let Ok(size) = env::var(size_id) {
        shm_size = Some(size.parse::<usize>().unwrap());
    }
    (ret, shm_size)
}

pub fn init_forkserver_fd(id: &str) -> Option<i32> {
    if let Ok(fd) = env::var(id) {
        return Some(fd.parse::<i32>().unwrap());
    }
    None
}
```

Initializing these file descriptors is straightforward.

As instrumented, the `__fuzzer_forkserver_init()` function sets up a loop that waits for a signal from the fuzzer to begin execution.

```rust
// fuzz_runtime/src/coverage.rs
pub const PROCESS_EXIT_NORMAL: u64 = 1;
pub fn read_wakeup(&self) {
    if let Some(fd) = self.fork_server_runtime {
        loop {
            let mut host_sent: u64 = 0;
            unsafe {
```

```
            read(
                fd,
                &mut host_sent as *mut _ as *mut c_void,
                mem::size_of::<u64>(),
            );
        }
        if host_sent == 99999999999 {
            // exit signal
            unsafe {
                libc::exit(0);
            }
        }
        let pid = unsafe { libc::fork() };
        if pid == 0 {
            // child process
            return; // run target program's main logic
        } else {
            // parent process
            let (tx, rx) = mpsc::channel();
            thread::spawn(
                move || match rx.recv_timeout(Duration::from_secs(host_sent)) {
                    Ok(_) => {}
                    Err(mpsc::RecvTimeoutError::Timeout) => unsafe {
                        kill(pid, SIGKILL);
                    },
                    _ => {}
                },
            );
            let mut status: i32 = 0;
            unsafe {
                waitpid(pid, &mut status, 0);
                tx.send(()).ok();
            }
            status = if status != SIGKILL {
                WEXITSTATUS(status)
            } else {
                status
            };
```

```rust
                self.notify_process_exit(status.try_into().unwrap());
            }
        }
    }
}
```

The `read_wakeup()` function enters an infinite loop, performing a blocking read to wait for a wakeup signal from the fuzzer. If the fuzzer sends the value `99999999999`, it indicates that the fuzzing process is complete, and the runtime can safely release resources. Otherwise, the received value is treated as a timeout, which sets a time limit for the target program's execution.

Once the signal is received, the runtime forks the target process using `libc::fork()`. The child process executes the target program's main logic using the provided fuzzing input (which fed from `stdin` or read from file), while the parent process waits for the child to finish. If the child process does not terminate within the specified timeout, it is forcibly killed.

```rust
// fuzz_runtime/src/coverage.rs
fn notify_process_exit(&self, status: u64) {
    if let Some(fd) = self.fork_server_host {
        // do not send zero value
        let status = if status == 0 {
            PROCESS_EXIT_NORMAL
        } else {
            status
        };
        let bytes = status.to_ne_bytes();
        unsafe { write(fd, bytes.as_ptr() as *const _,
        bytes.len()) };
    }
}
```

```rust
// fuzz_runtime/src/internal.rs
pub fn read_u64(mem: &[u8], start: usize) -> u64 {
    let mut slice = [0u8; 8];
    slice.copy_from_slice(&mem[start..start + 8]);
    u64::from_le_bytes(slice)
}

pub fn read_u128(mem: &[u8], start: usize) -> u128 {
    let mut slice = [0u8; 16];
    slice.copy_from_slice(&mem[start..start + 16]);
    u128::from_le_bytes(slice)
```

```
}

pub fn read_cov_report(mem: &[u8]) -> Vec<CovReport> {
    let size = read_u64(mem, 0) as usize;
    let mut slice = vec![0u8; size];
    slice.copy_from_slice(&mem[8..8 + size]);
    if let Ok(des) = bincode::deserialize(&slice) {
        des
    } else {
        vec![]
    }
}

pub fn write_u64(mem: &mut [u8], start: usize, v: usize) {
    mem[start..start + 8].copy_from_slice(&v.to_le_bytes())
}

pub fn write_u128(mem: &mut [u8], start: usize, v: u128) {
    mem[start..start + 16].copy_from_slice(&v.to_le_bytes())
}
```

Finally, the parent process checks the exit status of the child process and sends it back to the fuzzer. This status indicates whether the program terminated abnormally, allowing the fuzzer to determine whether a crash has occurred.

The second function is coverage measurment, which is handled by `trace_edge()`.

```rust
// fuzzer_runtime/src/coverage.rs

const PREV_LOC_IDX: usize = 0; // 16 byte [0..15]
pub const NEW_COVERAGES: usize = PREV_LOC_IDX + 16; // 8 byte [16..23]
pub const VISIT_EDGES: usize = NEW_COVERAGES + 8; // 8 bytes [24..31]
pub const VISIT_MARK: usize = VISIT_EDGES + 8; // 8 bytes [32..39]
pub const VISIT_EDGES_INDICIES: usize = VISIT_MARK + 8; // n bytes [40..]
pub const VISIT_EDGES_INDEX_SIZE: usize = 8;

pub fn trace_edge(&mut self, cur_loc: i64) {
    if let (Some(ref mut shm), Some(shm_size), Some(ref mut shm_aux)) =
        (&mut self.shm, self.shm_size, &mut self.aux)
    {
        let cur_loc = cur_loc as u128;
```

```
        let prev_loc = read_u128(shm_aux, PREV_LOC_IDX);
        let edge = (cur_loc ^ prev_loc) as usize % shm_size;
        write_u128(shm_aux, PREV_LOC_IDX, cur_loc >> 1);

        let visited_cnt = read_u64(shm_aux, VISIT_MARK) as usize;
        let already_visited = (0..visited_cnt).any(|i| {
            let stored_edge =
                read_u64(shm_aux, VISIT_EDGES_INDICIES + (i * VISIT_EDGES_INDEX_SIZE));
            stored_edge as usize == edge
        });
        if !already_visited {
            write_u64(
                shm_aux,
                VISIT_EDGES_INDICIES + (visited_cnt * VISIT_EDGES_INDEX_SIZE),
                edge,
            );
            write_u64(shm_aux, VISIT_MARK, visited_cnt + 1);
        }
        if shm[edge] == 0 {
            write_u64(shm_aux, NEW_COVERAGES, 1);
            let edges = read_u64(shm_aux, VISIT_EDGES) as usize;
            write_u64(shm_aux, VISIT_EDGES, edges + 1);
        }
        shm[edge] = shm[edge].saturating_add(1);
    }
}
```

We instrument a call to the `__fuzzer_trace_edge()` runtime function at every basic block. This function, in turn, calls `trace_edge` to process each executed edge.

The first step within `trace_edge()` is to calculate a unique edge index. To do this, we use `shm_aux` to store the previous basic block's location. This allows us to represent the transition (edge) between the previous and current block.

Next, we check if this specific edge has been visited before during the current execution. If it's a new edge for this run, we mark it as visited and increment a counter for the total number of unique edges visited. This information is crucial for the fuzzer, as it needs to know which edges were covered by the current seed and how frequently. This helps in seed evaluation through bucketing, a process where the fuzzer assesses the novelty and impact of each input. To accurately track per-run coverage, we record all visited edges for each target program invocation and clear this specific set of visited edges once the program exits. This ensures that for every new run, the fuzzer has a precise understanding of the edges covered by that specific input.

Finally, the overall visit counts for all edges are managed within `shm` (shared memory). It's critical to note that `shm` is not cleared after each program execution. This persistence allows the fuzzer to maintain a cumulative record of how frequently each edge has been hit across all fuzzing runs. If `shm` were cleared, the fuzzer would treat every edge as "new" in every single execution.

For example, imagine the fuzzer discovers a new code path when it processes Input A. Now, in the next run, the fuzzer mutates Input A to create Input B and feeds it to the program. If Input B happens to traverse the exact same code path as Input A, there's no "new" discovery in terms of unique paths for this particular run. **This is precisely why the edge count must persist across runs**. If it didn't, the fuzzer would repeatedly register already-found paths as "new," significantly hindering its ability to efficiently explore truly novel code.

5.4.3 Fuzzer

We've successfully implemented both the instrumentation module and its corresponding runtime handler. Our next and final step is to implement the fuzzer itself. Let's begin by examining its `main()` function.

5.4.3.1 Initialization

Rust

```rust
// fuzzer/src/mmap.rs
pub const SHM_PATH: &str = "/tmp/fuzzer_shared_mem";
pub const SHM_AUX_PATH: &str = "/tmp/fuzzer_shared_aux_mem";
pub const SHM_COV_PATH: &str = "/tmp/fuzzer_shared_cov_mem";
pub const SHM_SIZE: usize = 1 << 16;
pub const SHM_AUX_SIZE: usize = 1 << 20;
pub const SHM_COV_SIZE: usize = 1 << 13;

// fuzzer/src/main.rs
fn main() -> Result<()> {
    let (pgm_path, seed_dir_path, input_typ) = get_args()?;
    let init_seeds = SeedPool::new(&seed_dir_path);
    let (tx, rx) = mpsc::channel();
    let mut fuzzer = Fuzzer::new(
        SHM_PATH,
        SHM_SIZE,
        SHM_AUX_PATH,
        SHM_AUX_SIZE,
        SHM_COV_PATH,
        SHM_COV_SIZE,
        init_seeds,
        input_typ,
        tx,
    );
    let handle = std::thread::spawn(move || {
        let _ = run_ui(rx);
```

```
    });
    let _ = fuzzer.run(&pgm_path);
    handle.join().unwrap();
    Ok(())
}
```

The `main()` function initializes:

- CLI: Takes prgoram arguments
- SeedPool: Construct a seed pool object which is detailed later
- Fuzzer: Construct a fuzzer object
- UI: Configure fuzzer UI, which is detailed later
- Fuzzing campaign: Run fuzzing process

We won't include the command-line interface (CLI) implementation here; you can find it in the source code repository. The fuzzer's initialization primarily involves setting up shared memory.

```rust
// fuzzer/src/fuzzer.rs
pub struct Fuzzer {
    shm: SHM,
    shm_aux: SHM,
    shm_cov: SHM,
    pub forkserver_host: i32,
    pub forkserver_runtime: i32,
    seeds: SeedPool,
    new_paths: usize,
    input_typ: FuzzInput,
    seed_file_path: String, // if fuzz input is `ProgramArgument`, we write seed into this file path
    crashes: HashSet<Seed>,
    tx: mpsc::Sender<FuzzShot>,
    fuzz_terminate: bool,
}

impl Fuzzer {
    pub fn new(
        shm_path: &str,
        shm_size: usize,
        shm_aux_path: &str,
        shm_aux_size: usize,
        shm_cov_path: &str,
        shm_cov_size: usize,
        init_seeds: SeedPool,
        input_typ: FuzzInput,
```

```rust
        tx: mpsc::Sender<FuzzShot>,
    ) -> Self {
        let shm = SHM::new(shm_path, shm_size);
        let shm_aux = SHM::new(shm_aux_path, shm_aux_size);
        let shm_cov = SHM::new(shm_cov_path, shm_cov_size);
        let host_efd: RawFd = unsafe { eventfd(0, 0) };
        let runtime_efd: RawFd = unsafe { eventfd(0, 0) };
        Self {
            shm,
            shm_aux,
            shm_cov,
            forkserver_host: host_efd,
            forkserver_runtime: runtime_efd,
            seeds: init_seeds,
            new_paths: 0,
            input_typ,
            seed_file_path: format!("{}.seed", Uuid::new_v4()),
            crashes: HashSet::new(),
            tx,
            fuzz_terminate: false,
        }
    }
  }
```

Shared memory struct is defined as:

```rust
// fuzzer/src/mmap.rs
pub struct SHM {
    mem: MmapMut,
    mem_path: String,
    mem_size: usize,
}

impl SHM {
    pub fn new(shm_path: &str, shm_size: usize) -> Self {
        // 1. Create shared memory file
        let _ = fs::remove_file(shm_path); // clean up any previous file
        let file = OpenOptions::new()
            .read(true)
            .write(true)
            .create(true)
```

```
            .open(shm_path)
            .unwrap();
        file.set_len(shm_size as u64).unwrap();

        // 2. Memory-map the file
        let m = unsafe { MmapMut::map_mut(&file).unwrap() };
        let mut mmap = Self {
            mem: m,
            mem_path: shm_path.to_string(),
            mem_size: shm_size,
        };

        // 3. Zeroize
        mmap.zeroize();
        mmap
    }

    pub fn mut_mem(&mut self) -> &mut MmapMut {
        &mut self.mem
    }

    pub fn mem(&self) -> &MmapMut {
        &self.mem
    }

    pub fn path(&self) -> &str {
        &self.mem_path
    }

    pub fn size(&self) -> usize {
        self.mem_size
    }

    pub fn zeroize(&mut self) {
        self.mem[..].fill(0);
    }
}
```

5.4.3.2 Seed Initialization

There are two structures: `Seed` and `SeedPool`.

The struct `Seed` contains two important data:

```rust
#[derive(Debug, Clone)]
pub struct Seed {
    input: Vec<u8>,
    score: u64,
}
```

The input represents the seed data, and score is its evaluated value, determined by the fuzzer.

Seeds are managed by the `SeedPool` struct. This struct internally utilizes a `BTreeSet`, which means any type used within it must implement the `Hash` trait. Additionally, the `mutate()` function is available to perform modifications on a given seed.

The `SeedPool` offers two primary methods: `add_seed()` to incorporate new seeds, and `pop_seed()` to retrieve the most promising seed from the pool.

```rust
// fuzzer/src/seed.rs
const MAX_SEED_LEN: usize = 1000;
const CRASH_OUTPUT_DIR: &str = "crashes";

#[derive(Debug, Clone)]
pub struct Seed {
    input: Vec<u8>,
    score: u64,
}

impl Seed {
    pub fn new(input: Vec<u8>, score: u64) -> Self {
        Self { input, score }
    }

    pub fn mutate(&mut self) -> Vec<MutateResult> {
        mutator::mutate(&mut self.input)
    }

    pub fn get_input(&self) -> &[u8] {
        &self.input
    }

    pub fn get_score(&self) -> u64 {
        self.score
    }

    pub fn to_hex(&self) -> String {
        self.get_input()
```

```
                .iter()
                .map(|b| format!("{:02X}", b))
                .collect::<Vec<_>>()
                .join("")
        }

        pub fn to_file(&self, cnt: usize) {
            fs::create_dir_all(CRASH_OUTPUT_DIR).unwrap();
            let mut file_path = PathBuf::from(CRASH_OUTPUT_DIR);
            file_path.push(format!("{}.crash", cnt));
            fs::write(&file_path, self.input.clone()).unwrap();
        }

        pub fn set_score(&mut self, v: u64) {
            self.score = v;
        }
    }

    impl Ord for Seed {
        fn cmp(&self, other: &Self) -> std::cmp::Ordering {
            self.score
                .cmp(&other.score)
                .then_with(|| self.input.cmp(&other.input))
        }
    }

    impl PartialOrd for Seed {
        fn partial_cmp(&self, other: &Self) ->
        Option<std::cmp::Ordering> {
            Some(self.cmp(other))
        }
    }

    impl PartialEq for Seed {
        fn eq(&self, other: &Self) -> bool {
            self.input == other.input && self.score == other.score
        }
    }

    impl Eq for Seed {}

```

```rust
impl Hash for Seed {
    fn hash<H: Hasher>(&self, state: &mut H) {
        self.input.hash(state);
        self.score.hash(state);
    }
}

pub struct SeedPool {
    seeds: BTreeSet<Seed>,
}

impl SeedPool {
    pub fn new(seed_dir: &str) -> Self {
        let mut btree = BTreeSet::new();
        let init_seeds = read_seed_dir(seed_dir).unwrap();
        for init_seed in init_seeds {
            btree.insert(Seed::new(init_seed, 0));
        }
        Self { seeds: btree }
    }

    pub fn add_seed(&mut self, seed: Seed) {
        if self.seeds.len() > MAX_SEED_LEN {
            if let Some(min_seed) = self.get_min_score_seed() {
                self.seeds.remove(&min_seed);
            }
        }
        self.seeds.insert(seed);
    }

    fn get_min_score_seed(&self) -> Option<Seed> {
        self.seeds.iter().next().cloned()
    }

    fn get_max_score_seed(&self) -> Option<Seed> {
        self.seeds.iter().next_back().cloned()
    }

    pub fn pop_seed(&mut self) -> Option<Seed> {
        if let Some(seed) = self.get_max_score_seed() {
            self.seeds.remove(&seed);
```

```
            Some(seed)
        } else {
            None
        }
    }

    pub fn is_empty(&self) -> bool {
        self.seeds.len() == 0
    }

    pub fn len(&self) -> usize {
        self.seeds.len()
    }
}
```

5.4.3.3 Target Program Initialization

Initially, the target program is set up using the `Command` crate. During this setup, several environment variables are notably configured.

- `LD_LIBRARY_PATH`: Sets current directory as library path, meaning that fuzzer runtime should locate within current directory
- `SHM_XXX`: Shared memory-releated values such as file path
- `FORK_SERVER_XXX`: Event notification related values

It's worth noting that `stdout` and `stderr` are piped. This allows us to intercept the target program's terminal output and render these outputs directly within our fuzzer UI.

```rust
// fuzzer/src/fuzzer.rs
pub fn run(&mut self, program_file: &str) -> Result<FuzzResult> {
    let mut cmd = Command::new(program_file);
    let child_process_cmd = cmd
        .env("LD_LIBRARY_PATH", ".")
        .env("SHM_ID", self.shm.path())
        .env("SHM_AUX_ID", self.shm_aux.path())
        .env("SHM_SIZE", format!("{}", &self.shm.size()))
        .env("SHM_AUX_SIZE", format!("{}", &self.shm_aux.size()))
        .env("SHM_COV_ID", self.shm_cov.path())
        .env("SHM_COV_SIZE", format!("{}", &self.shm_cov.size()))
        .env("FORK_SERVER_HOST", format!("{}", self.forkserver_host))
        .env(
            "FORK_SERVER_RUNTIME",
            format!("{}", self.forkserver_runtime),
        )
        .stdout(Stdio::piped())
```

```
18          .stderr(Stdio::piped());
19
20      if self.input_typ == FuzzInput::ProgramArgument {
21          child_process_cmd.args(vec![&self.seed_file_path]);
22      } else {
23          child_process_cmd.stdin(Stdio::piped());
24      }
25
26      let mut child_process = child_process_cmd.spawn()?;
27      let mut child_stdin = if self.input_typ == FuzzInput::Stdin {
28          Some(child_process.stdin.take().unwrap())
29      } else {
30          None
31      };
32
33      ...
34      ...
35
36  }
```

At line 26, we load the target program, which will then enter a blocked state.

5.4.3.4 Harness

As previously explained, the target program needs to implement a harness to interact with the fuzzer. In this book, we'll use two methods for this interaction: `stdin` and file.

If `stdin` is chosen, we intercept the input at lines 27-29. The `feed_seed()` function then determines which harness is in use. If it's `stdin`, the function writes a seed directly to the `stdin` file descriptor; otherwise, it writes the seed into a file.

```rust
fn feed_seed(&self, child_stdin: &mut Option<ChildStdin>,
seed: &Seed) -> Result<()> {
    if let Some(ref mut stdin) = child_stdin {
        stdin.write_all(seed.get_input())?;
    }
    if self.input_typ == FuzzInput::ProgramArgument {
        write_seed(&self.seed_file_path, seed.get_input())?;
    }
    Ok(())
}
```

5.4.3.5 Fuzzing Campaign

The entire fuzzing process is attached below:

```rust
  pub fn run(&mut self, program_file: &str) ->
  Result<FuzzResult> {
```

```
        let mut cmd = Command::new(program_file);
        let child_process_cmd = cmd
            .env("LD_LIBRARY_PATH", ".")
            .env("SHM_ID", self.shm.path())
            .env("SHM_AUX_ID", self.shm_aux.path())
            .env("SHM_SIZE", format!("{}", &self.shm.size()))
            .env("SHM_AUX_SIZE", format!("{}", &self.shm_aux.size()))
            .env("SHM_COV_ID", self.shm_cov.path())
            .env("SHM_COV_SIZE", format!("{}", &self.shm_cov.size()))
            .env("FORK_SERVER_HOST", format!("{}",
            self.forkserver_host))
            .env(
                "FORK_SERVER_RUNTIME",
                format!("{}", self.forkserver_runtime),
            )
            .stdout(Stdio::piped())
            .stderr(Stdio::piped());

        if self.input_typ == FuzzInput::ProgramArgument {
            child_process_cmd.args(vec![&self.seed_file_path]);
        } else {
            child_process_cmd.stdin(Stdio::piped());
        }

        let mut child_process = child_process_cmd.spawn()?;
        let mut child_stdin = if self.input_typ == FuzzInput::Stdin {
            Some(child_process.stdin.take().unwrap())
        } else {
            None
        };

        self.pgm_output_reader(
            child_process.stdout.take().unwrap(),
            child_process.stderr.take().unwrap(),
        );
        let mut init_set_timeout = false;
        let mut timeout = Duration::new(9999, 0);
        let fuzzer_started = Instant::now();

        let mut loop_cnt = 0;
        if self.is_seed_empty() {
```

```
            return Ok(FuzzResult::AllSeedConsumed(loop_cnt));
        }
        let mut seed = self.pop_seed().unwrap();

        loop {
            if self.fuzz_terminate {
                self.terminate();
                self.send(FuzzShot::Terminated);
                return Ok(FuzzResult::UserTerminated);
            }
            loop_cnt += 1;
            self.feed_seed(&mut child_stdin, &seed)?;
            self.wakeup_forkserver(HostSend::Wakeup(timeout.as_secs()));
            let target_started = Instant::now();
            let status = self.wait_forkserver();

            let elapsed = target_started.elapsed();
            if !init_set_timeout {
                timeout = elapsed * INITIAL_TIMEOUT_UPPER_BOUND;
                init_set_timeout = true;
            }
            // evaluate seed and add it
            let (visit_edges, score) = self.eval_seed(status);
            self.debug(
                loop_cnt,
                &seed,
                score,
                timeout,
                elapsed,
                fuzzer_started.elapsed(),
            );
            self.clear_new_coverage();
            self.clear_visited_edges();
            if self.is_seed_empty() {
                self.terminate();
                self.send(FuzzShot::Terminated);
                return Ok(FuzzResult::AllSeedConsumed(loop_cnt));
            }

            // crash found
```

```
82             if self.is_crash(status) {
83                 if let Ok(minimized) = self.ddmin(&mut child_stdin, timeout, &seed) {
84                     self.crashes.insert(minimized.clone());
85                     minimized.to_file(self.crashes.len());
86                     self.send(FuzzShot::Crash(CrashInfo::new(
87                         self.crashes.len(),
88                         seed,
89                         minimized,
90                     )));
91                 }
92             } else if score > 0 {
93                 // re-evaulate score after execution
94                 seed.set_score(score);
95                 self.add_seed(seed.clone());
96                 let cur_seed = seed.clone();
97                 seed.mutate();
98                 seed.set_score(score.saturating_add(1));
99                 self.add_seed(seed.clone());
100
101                self.send(FuzzShot::SeedInfo(FuzzerSeed::new(
102                    self.seeds.len(),
103                    cur_seed,
104                    seed,
105                    visit_edges,
106                    self.new_paths,
107                )));
108            }
109            seed = self.pop_seed().unwrap();
110        }
111    }
```

At lines 36-38, we initialize the timeout and record the fuzzer's start time. One seed is popped from the seed pool between lines 41 and 44. The main fuzzing loop begins at line 46. If the fuzzer should terminate, it exits the function at lines 47-51. This termination condition becomes true when the UI is exited (e.g., by typing q), a signal handled by the UI implementation.

We feed a seed input at line 53 and then send a wakeup signal to the fuzzer runtime at line 54. Following this, the target program will fork and execute its main logic. The fuzzing input will be delivered via the harness.

After the target program begins execution, the fuzzer waits for its completion. Once the target program finishes, it sends its exit status back to the fuzzer.

```rust
// fuzzer/src/fuzzer.rs
```

```rust
pub fn wait_forkserver(&mut self) -> i32 {
    let mut status: u64 = 0;
    unsafe {
        read(
            self.forkserver_host,
            &mut status as *mut _ as *mut c_void,
            mem::size_of::<u64>(),
        );
    }
    status as i32
}
```

At line 64, the given seed is evaluated, producing a score value. If new coverage is discovered, the seed receives a higher score; otherwise, its score is determined through a process called bucketing.

```rust
// fuzzer/src/fuzzer.rs
pub fn eval_seed(&mut self, status: i32) -> (u64, u64) {
    if status == SIGKILL {
        return (0, 0); // give zero score for hangs
    }
    let visit_edges = read_u64(self.shm_aux.mem(), VISIT_MARK);
    // if new coverage found, give max score, otherwise give higher
    score if the path is rare
    if self.is_new_coverage() {
        self.new_paths += 1;
        return (visit_edges, u64::MAX);
    }
    let mut score = 0;
    for i in 0..visit_edges {
        let edge = read_u64(
            self.shm_aux.mem(),
            VISIT_EDGES_INDICIES + (i as usize) *
            VISIT_EDGES_INDEX_SIZE,
        ) as usize;
        if edge != 0 {
            score += get_score(self.shm.mem()[edge]) as u64;
        }
    }
    (visit_edges, score)
}
```

Bucketing is implemented as follows:

Rust
```rust
// fuzzer/src/bucket.rs
const BUCKET_MAX_VALUE: u8 = 8;
const COUNT_CLASS_LOOKUP: [u8; 256] = {
    let mut table = [0u8; 256];
    let mut i = 0;
    while i < 256 {
        table[i] = match i {
            0..=1 => 1,
            2 => 2,
            3 => 3,
            4..=7 => 4,
            8..=15 => 5,
            16..=31 => 6,
            32..=127 => 7,
            _ => BUCKET_MAX_VALUE,
        };
        i += 1;
    }
    table
};

/// lower value is more valuable because it has been not rarely exercised
fn to_bucket(value: u8) -> u8 {
    COUNT_CLASS_LOOKUP[value as usize]
}

pub fn get_score(value: u8) -> u8 {
    let score = to_bucket(value);
    (BUCKET_MAX_VALUE + 1) - score
}
```

As described earlier, a lower hit count is more valuable, while a higher hit count will be assigned a lower score.

Once the score evaluation is complete, the coverage footprint — specifically, the number of visited edges, which edges are new, and which edges have been visited — must be cleared. This information can be reset via the `shm_aux` shared memory.

Rust
```rust
// fuzzer/src/fuzzer.rs
pub fn clear_new_coverage(&mut self) {
    write_u64(self.shm_aux.mut_mem(), NEW_COVERAGES, 0);
}

```

```rust
pub fn clear_visited_edges(&mut self) {
    let visit_edges = read_u64(self.shm_aux.mem(), VISIT_MARK);
    for i in 0..visit_edges {
        write_u64(
            self.shm_aux.mut_mem(),
            VISIT_EDGES_INDICIES + (i as usize) * VISIT_EDGES_INDEX_SIZE,
            0,
        );
    }
    write_u64(self.shm_aux.mut_mem(), VISIT_MARK, 0);
}
```

If the seed pool becomes empty, the fuzzer halts, as shown at lines 75-79. At this point, it also sends a terminate event to the runtime.

Should the current seed lead to a crash, the fuzzer will minimize the input and store it to disk (lines 82-91).

Finally, we reset the evaluated score for the current seed and add it back into the pool. We also create a copy of this seed, mutate it, and then add the mutated version to the pool. Before the loop restarts, a promising seed is selected for the next execution (line 92 - 109).

5.4.3.6 Mutation

Up to this point, we've covered the fuzzer's entire workflow, but we haven't yet discussed how new seeds are generated. Our approach is to create new seeds by mutating the existing seeds. In this book, we will focus on five widely used and effective mutation methods.

- Insertion: Inserts one random byte at a random position within the seed
- Deletion: Removes a single random byte from the seed
- Change: Replaces one random byte in the seed with a new random byte
- Flip: Flips the bits of a single random byte (e.g., `seed[idx] ^= 0xFF`)
- Arithmetic: Adds a randomly generated integer (within the range of -35 to $+35$, as used by AFL) to a random position in the seed

The five methods were implemented as follows:

```rust
// fuzzer/src/mutator.rs

use rand::Rng;

// 1. random insert
// 2. random delete
// e. random change
// 4. random flip
// 5. arithmetic mutation

```

```
const KEEP_SEED_MIN_LEN: usize = 1;

#[derive(PartialEq, Eq, Debug)]
pub enum MutateResult {
    EmptyInput,
    SeedTooShortToDelete,
    SeedTooShortToArithmeticMutate,
    Done,
}

fn gen_new_byte() -> u8 {
    rand::random::<u8>()
}

fn gen_random_idx(len: usize) -> usize {
    rand::random::<usize>() % len
}

type MutatorFn = fn(&mut Vec<u8>) -> MutateResult;

const MUTATORS: &[MutatorFn] = &[insert, change, delete, flip, arithmetic];

fn insert(seed: &mut Vec<u8>) -> MutateResult {
    if seed.is_empty() {
        return MutateResult::EmptyInput;
    }
    let idx = gen_random_idx(seed.len());
    seed.insert(idx, gen_new_byte());
    MutateResult::Done
}

fn change(seed: &mut Vec<u8>) -> MutateResult {
    if seed.is_empty() {
        return MutateResult::EmptyInput;
    }
    let idx = gen_random_idx(seed.len());
    seed[idx] = gen_new_byte();
    MutateResult::Done
}

```

```
fn delete(seed: &mut Vec<u8>) -> MutateResult {
    if seed.is_empty() {
        return MutateResult::EmptyInput;
    }
    let idx = gen_random_idx(seed.len());
    if seed.len() > KEEP_SEED_MIN_LEN {
        seed.remove(idx);
        MutateResult::Done
    } else {
        MutateResult::SeedTooShortToDelete
    }
}

fn flip(seed: &mut Vec<u8>) -> MutateResult {
    if seed.is_empty() {
        return MutateResult::EmptyInput;
    }
    let idx = gen_random_idx(seed.len());
    seed[idx] ^= 0xFF;
    MutateResult::Done
}

fn arithmetic(seed: &mut Vec<u8>) -> MutateResult {
    if seed.is_empty() {
        return MutateResult::EmptyInput;
    }
    let mut rng = rand::thread_rng();
    let mutate_widths = [1, 2, 4];
    let width = match seed.len() {
        0..=1 => return MutateResult::SeedTooShortToArithmeticMutate,
        2..=3 => 1,
        4 => mutate_widths[rng.gen_range(0..2)],
        _ => mutate_widths[rng.gen_range(0..mutate_widths.len())],
    };
    let idx = rng.gen_range(0..=seed.len() - width);
    // choose a random value from -35 ~ +35
    let delta = rng.gen_range(-35i64..=35);
    match width {
        1 => {
            let val = seed[idx];
```

```
            let new_val = val.wrapping_add(delta as u8);
            seed[idx] = new_val;
        }
        2 => {
            let val = u16::from_le_bytes([seed[idx], seed[idx + 1]]);
            let new_val = val.wrapping_add(delta as u16);
            let bytes = new_val.to_le_bytes();
            seed[idx..idx + 2].copy_from_slice(&bytes);
        }
        4 => {
            let val = u32::from_le_bytes([seed[idx], seed[idx + 1], seed[idx + 2], seed[idx + 3]]);
            let new_val = val.wrapping_add(delta as u32);
            let bytes = new_val.to_le_bytes();
            seed[idx..idx + 4].copy_from_slice(&bytes);
        }
        _ => unreachable!(),
    }
    MutateResult::Done
}

pub fn mutate(seed: &mut Vec<u8>) -> Vec<MutateResult> {
    // muate 1 ~ 10% for a given seed
    let mut rng = rand::thread_rng();
    let max = ((seed.len() as f32) * 0.2).ceil() as usize;
    let max = max.max(1).min(seed.len());
    let n = rng.gen_range(1..=max);

    let mut results = vec![];
    for _ in 0..=n {
        let idx = gen_random_idx(MUTATORS.len());
        let result = MUTATORS[idx](seed);
        results.push(result);
    }
    results
}
```

Each mutator is chosen randomly, and the degree of mutation has been predetermined for this book. We apply mutations ranging from 1% to 10% per seed, meaning each seed will be altered by at most 10% of its original content.

5.4.3.7 UI

From a functional standpoint, everything is now implemented. You'll soon see the fuzzer's user interface, which is our last piece of the puzzle: wrapping it all with an intuitive and awesome UI. Our intended UI layers are as follows:

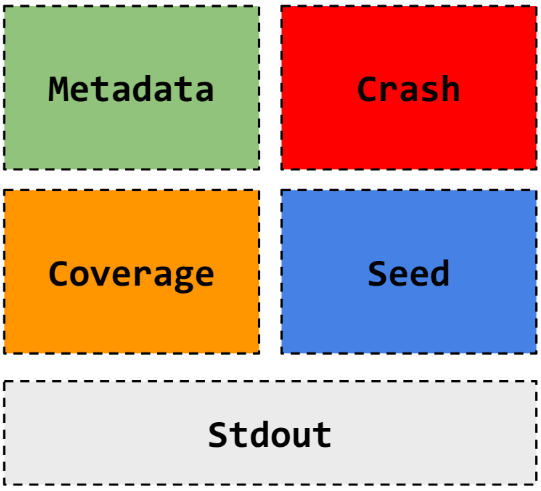


Figure 5.5: Implementation Components

There are five compartments.

- Metadata
  - Fuzzing loop counter: Increased if target program is ran once
  - Fuzzing input type: `<stdin | file>`
  - Fuzzing Timeout: Time constraint of target program invocation
  - Elapsed time of single run: An elapsed time of target program invocation
  - Uptime: Fuzzer's uptime
- Crash
  - Crahses: Number of found crashes
  - Origin: An original bug-triggering seed
  - Origin length: Length of `Origin`
  - Minimized: Minimized `Origin`
  - Minimized length: Length of `Minimized`
- Coverage
  - File: Name of target program file
  - Function hit ratio: Function hit ratio from coverage module (chapter 1)
  - Branch hit ratio: Branch hit ratio from coverage module (chapter 1)
  - Line hit ratio: Line hit ratio from coverage module (chapter 1)
- Seed
  - Seeds: Number of generated seeds
  - Current seed: A current seed to be fed
  - Current seed score: Score of `Current seed`
  - Visited edges: Visited number of basic blocks
  - Next seed: A seed to be fed at next round
  - New paths: Number of found new edges.
- Stdout
  - Display the `stdout` of target program

Note that the `Coverage` panel will not display live numbers unless the coverage instrumentation and its corresponding runtime are actively involved. The following function defines five distinct data structures, one for each panel.

```rust
// fuzzer/src/campaign.rs

#[derive(Debug)]
pub struct CrashInfo {
    pub crashes: usize,
    pub origin: Seed,
    pub minimized: Seed,
}
impl CrashInfo {
    pub fn new(crashes: usize, origin: Seed, minimized: Seed) -> Self {
        return Self {
            crashes,
            origin,
            minimized,
        };
    }
}

#[derive(Debug)]
pub struct FuzzerMetadata {
    pub fuzz_cnt: u64,
    pub fuzz_input_typ: FuzzInput,
    pub timeout: Duration,
    pub target_elapsed_time: Duration,
    pub total_elapsed_time: Duration,
}
impl FuzzerMetadata {
    pub fn new(
        fuzz_cnt: u64,
        fuzz_input_typ: FuzzInput,
        timeout: Duration,
        target_elapsed_time: Duration,
        total_elapsed_time: Duration,
    ) -> Self {
        Self {
            fuzz_cnt,
            fuzz_input_typ,
```

```
                timeout,
                target_elpased_time,
                total_elpased_time,
            }
        }
    }

    #[derive(Debug)]
    pub struct FuzzerSeed {
        pub seeds: usize,
        pub cur_seed: Seed,
        pub next_seed: Seed,
        pub visit_edges: u64,
        pub new_paths: usize,
    }
    impl FuzzerSeed {
        pub fn new(
            seeds: usize,
            cur_seed: Seed,
            next_seed: Seed,
            visit_edges: u64,
            new_paths: usize,
        ) -> Self {
            Self {
                seeds,
                cur_seed,
                next_seed,
                visit_edges,
                new_paths,
            }
        }
    }

    #[derive(Debug)]
    pub enum FuzzShot {
        ProgramOutput(String),
        Coverage(Vec<CovReport>),
        Crash(CrashInfo),
        Metadata(FuzzerMetadata),
        SeedInfo(FuzzerSeed),
        Terminated,
```

```rust
}

#[derive(Debug)]
pub struct FuzzingCampaign {
    pub program_output: Vec<String>,
    pub coverage: Vec<CovReport>,
    pub crash: CrashInfo,
    pub metadata: FuzzerMetadata,
    pub seed_info: FuzzerSeed,
}
impl FuzzingCampaign {
    pub fn new(fuzz_input: FuzzInput) -> Self {
        Self {
            program_output: vec!["".to_string()],
            coverage: vec![CovReport {
                file: "".to_string(),
                funcs_hit_ratio: "".to_string(),
                uncovered_funcs: "".to_string(),
                brs_hit_ratio: "".to_string(),
                uncovered_brs: "".to_string(),
                lines_hit_ratio: "".to_string(),
                uncovered_lines: "".to_string(),
            }],
            crash: CrashInfo::new(0, Seed::new(vec![], 0),
            Seed::new(vec![], 0)),
            metadata: FuzzerMetadata::new(
                0,
                fuzz_input,
                Duration::default(),
                Duration::default(),
                Duration::default(),
            ),
            seed_info: FuzzerSeed::new(0, Seed::new(vec![], 0),
            Seed::new(vec![], 0), 0, 0),
        }
    }
    pub fn add_program_output(&mut self, pgm_output: String) {
        self.program_output.push(pgm_output);
    }
    pub fn set_coverage(&mut self, coverage: Vec<CovReport>) {
        if !coverage.is_empty() {
            self.coverage = coverage;
```

```
            }
        }
        pub fn set_crash(&mut self, crash: CrashInfo) {
            self.crash = crash;
        }
        pub fn set_metadata(&mut self, metadata: FuzzerMetadata) {
            self.metadata = metadata;
        }
        pub fn set_seed_info(&mut self, seed_info: FuzzerSeed) {
            self.seed_info = seed_info;
        }
    }
```

The fuzzer sends these objects to the UI application, which then updates the live interface. Instead of presenting the entire code here, as it exceeds 300 lines, we'll focus only on the core logic.

```rust
// fuzzer/src/ui.rs
pub fn run_ui(rx: mpsc::Receiver<FuzzShot>) -> io::Result<()> {
    enable_raw_mode()?;
    let mut stdout = stdout();
    execute!(stdout, EnterAlternateScreen)?;
    let backend = CrosstermBackend::new(stdout);
    let mut terminal = Terminal::new(backend)?;

    let res = run_app(&mut terminal, rx)?;

    disable_raw_mode()?;
    execute!(terminal.backend_mut(), LeaveAlternateScreen)?;
    terminal.show_cursor()?;
    println!("{}", res);
    Ok(())
}

fn run_app<B: ratatui::backend::Backend>(
    terminal: &mut Terminal<B>,
    rx: mpsc::Receiver<FuzzShot>,
) -> io::Result<String> {
    let mut fuzz_result = FuzzingCampaign::new(FuzzInput::Stdin);
    loop {
        match rx.recv() {
            Ok(result) => match result {
```

```
26                 FuzzShot::ProgramOutput(o) => {
27                     fuzz_result.add_program_output(o);
28                 }
29                 FuzzShot::Coverage(cov) => {
30                     fuzz_result.set_coverage(cov);
31                 }
32                 FuzzShot::Crash(crash) => {
33                     fuzz_result.set_crash(crash);
34                 }
35                 FuzzShot::Metadata(md) => {
36                     fuzz_result.set_metadata(md);
37                 }
38                 FuzzShot::SeedInfo(seed) => {
39                     fuzz_result.set_seed_info(seed);
40                 }
41                 FuzzShot::Terminated => {
42                     return Ok("Fuzzer terminated".to_string());
43                 }
44             },
45             Err(err) => {
46                 panic!("{:?}", err);
47             }
48         }
49
50         terminal.draw(|frame| render_ui(frame, &mut fuzz_result))?;
51
52         let timeout = Duration::from_millis(10);
53         if event::poll(timeout)? {
54             if let Event::Key(key) = event::read()? {
55                 if KeyCode::Char('q') == key.code {
56                     return Ok("'Q' entered".to_string());
57                 }
58             }
59         }
60     }
61 }
```

The loop at line 23 receives live fuzzing data from the fuzzer, which then updates the UI at line 50. If the user presses ‘q’, the fuzzer terminates, the quit handler is presented at lines 52-60.

5.4.3.8 Fuzz run

With all implementations complete, we can now run the fuzzer in two steps: instrumentation and fuzzer execution.

Here's the example target program we'll be fuzzing:

```c
1  // target.c
2  #include <stdio.h>
3
4  #define FUZZ_LEN 10
5  #define ABNORMAL_EXIT 100
6
7  int main(int argc, char* argv[]) {
8      char input[FUZZ_LEN];
9      fgets(input, sizeof(input), stdin);
10     printf("target run\n");
11
12     if (input[0] == 'a') {
13         if (input[1] == 'c' || input[1] == 'd' || input[1] == 'e' ||
           input[1] == 'f') { // buggy path
14             // input[FUZZ_LEN] = 'a'; // you can use to trigger a
               explicit bug using OOB
15             return ABNORMAL_EXIT;
16         }
17     }
18     return 0;
19 }
```

This program accepts user input via the `stdin` file descriptor. We've intentionally introduced a bug at either line 14 or 15 (you can choose which one). This bug triggers if the input seed's first byte is 'a', and the `stdin` input also contains 'c', 'd', 'e', or 'f' at index 1.

Since we've implemented ASan, we can trigger an explicit OOB access to cause a crash. Alternatively, we can make the program return any value other than 0 or 1 to signal an error.

These are the compile commands:

```
clang -g -O0 -S -emit-llvm target.c -o target.ll
./instrument -i target.ll -o itarget.ll -m all
clang itarget.ll -L. -lcoverage_runtime -lfuzzer_runtime -
lasan_runtime
// the binary 'a.out' will be generated.
```

You'll need to generate your initial seed files within the `init-seeds` directory. For demonstration, we've simply added three seed files: `a.txt`, `b.txt`, and `c.txt`.

```
init-seeds/a.txt: aaba
init-seeds/b.txt: aaab
```

```
init-seeds/c.txt: abaa
```

Now we can run the fuzzer with the following command: `./fuzzer -p ./a.out -c init-seeds -t stdin`.

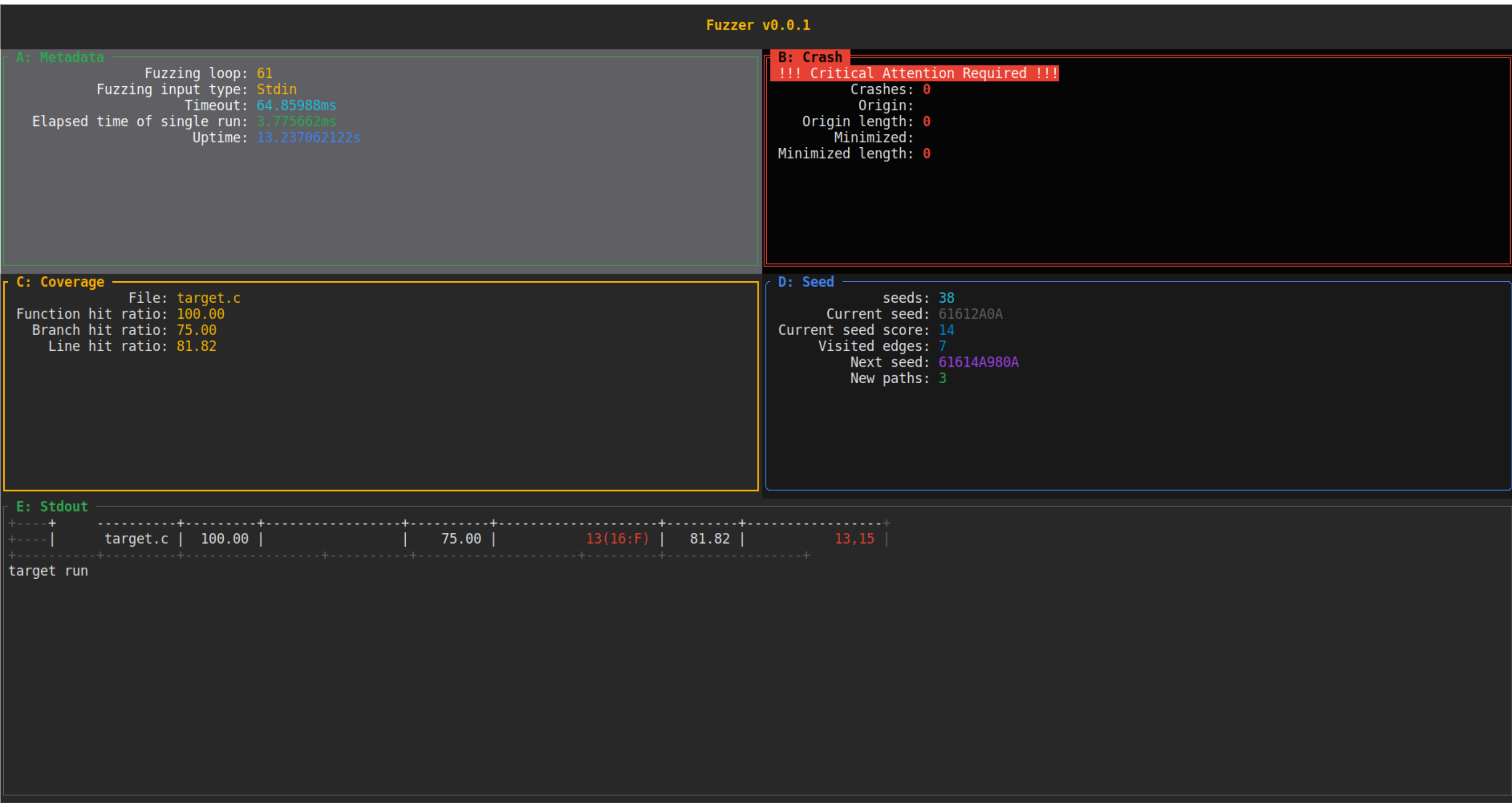


Figure 5.6: Fuzzer Execution

Crashes will be stored in the `crashes` directory, which is generated automatically.

```
crashes/1.crash: ae
crashes/2.crash: ad
crashes/3.crash: ac
```

The fuzzer successfully finds a buggy path for this example target program in under a minute on average. However, if you change the condition at line 13 to `input[1] == 'c'`, it'll take more time to trigger the bug. This is because the probability of generating a seed that satisfies this new condition is significantly lower (.because previously, the value at index 1 could be 'c', 'd', 'e', or 'f'. Now, it specifically requires 'c', ignoring 'd', 'e', and 'f'.).

5.4.4 Summary

We fully implemented AFL-like fuzzing campaign by designing and implementing instrumentation module, runtime, and fuzzer.

5.4.4.1 Homework

Exercise 5.1 We suggest the following topics for further study and exploration:

- **Topic #1: Comparing Vanilla Fuzzer vs. Fork-Server Based Fuzzer Performance**
  - You can experiment with a vanilla fuzzer that doesn't utilize the fork-server mechanism (i.e., `execve()` is called at every run), loading the target program at each fuzzing round. This will allow you to learn how much a fork-server based fuzzer contributes to boosting fuzzing speed.
- **Topic #2: Smart Mutation Strategy**
  - Currently, all mutation methods are selected randomly. However, you could assign weights to each method, giving higher probabilities to certain techniques under specific circumstances. By employing a heuristic approach to develop a smarter strategy manager, you could then compare its performance to a random selection and evaluate if it contributes to finding bugs more quickly.

Numerous highly-tuned fuzzers are regularly submitted to and published in Computer Science conferences. You are encouraged to explore these publications to discover interesting topics and study current trends and challenges in the field. We hope this area inspires you to pursue rewarding research and coding endeavors.

Part V
Symbolic Execution

“How to satisfy the branch condition to proceed?”

```
if (a + 5 == 42) {
  ... // Block 1
} else {
  ... // Block 2
}
```

```
Block 1: a == 37
Block 2: a != 37
```

6. Introduction to Symbolic Execution

6.1 Problem

In the previous chapter, we implemented a coverage-based fuzzer. The goal of this fuzzer is to determine the fuzzing direction through its coverage footprint. It’s important to understand that while it uses coverage to guide its decisions, it doesn’t inherently increase coverage itself. To truly boost the coverage ratio, a fuzzer needs to be aware of how to satisfy all branches so it can enter them, thereby increasing code coverage. For example, the branch condition below would be very difficult to satisfy via mutation alone.

```
1 if (input() == "0x9e6d3c81b42a7f124e5f9a51c23df61d5a97a4c3f7e9b881b3e4d2a6b92cbbaf") {
2   ...
3 } else {
4   ...
5 }
```

Most coverage-based fuzzers will never reach line 2, consistently taking only the `else` branch in every run. This is because they lack any hint or mechanism for generating inputs that could satisfy the conditions needed to reach other branches.

Let’s consider a simpler example.

```
if (input() > 100) {
  ...
} else {
  ...
}
```

To take the true branch, the `input()` value must be greater than 100. A coverage-based fuzzer might discover a value exceeding 100 through mutation, but this is purely a matter of random chance. This lack of awareness regarding branch conditions is a significant hurdle when trying to generate appropriate seed values for specific target branches.

Symbolic execution addresses this challenge by providing the ability to find a solution that satisfies a target branch. In this book, we won't be implementing a hybrid fuzzer that combines coverage-based techniques (from Chapter 3) with symbolic execution (from Chapter 4). This will be detailed in the homework exercises at the end of this chapter.

6.2 Working Example

To interact with symbolic execution, the programmer must symbolize specific variables. The values for these variables are then provided by the symbolic execution runtime. The following code is an example of symbolization.

```c
// symbolic-example.c
#include <stdio.h>

extern void __make_symbolic();

int main() {
    int i,j;
    __make_symbolic(sizeof(int), &i);
    __make_symbolic(sizeof(int), &j);
    printf("%d, %d\n", i,j);
    if (i == 0) {
        if (j == 123214125) {
            printf("S1\n");
        } else {
            printf("S2\n");
        }
    } else {
        if (j != 88148128) {
            printf("S3\n");
        } else {
            printf("S4\n");
            return 123;
        }
    }
}
```

The example target program contains six branches, and it evaluates and compares the symbolic variables `i` and `j`. The symbolic execution runtime provides the values for these variables. Here are the compilation commands. Note that we're using the `-m all` option, which enables all instrumentation modules.

```
clang -g -O0 -S -emit-llvm symbolic-example.c  -o symbolic-example.ll
./instrument -i symbolic-example.ll  -o isymbolic-example.ll -m all
```

```
clang isymbolic-example.ll -L. -lcoverage_runtime -lasan_runtime -
lfuzzer_runtime -lsymbolic_runtime
```

Let's run the generated binary.

```
❯ LD_LIBRARY_PATH=./ ./a.out
1, 123214126
S3
... (coverage report)

❯ LD_LIBRARY_PATH=./ ./a.out
1, 88148128
S4
... (coverage report)

❯ LD_LIBRARY_PATH=./ ./a.out
0, 123214125
S1
... (coverage report)

❯ LD_LIBRARY_PATH=./ ./a.out
0, 88148128
S2
... (coverage report)
```

Even after multiple runs, all branches are covered. However, with a coverage-based fuzzer, generating specific values like `i == 0 && j = 123214125`(printing "S1") or `i == 1 && j = 88148128`(printing "S4") would be extremely difficult. While the final binary can be integrated with our coverage-based fuzzer (`./fuzzer -p ./a.out -c init-seeds -t stdin`), the fuzzer will essentially just repeat the execution process. This means the symbolic values will still be provided by the symbolic execution runtime, and the fuzzer itself won't be aware of the symbolic execution process. A detailed look at the ideal integration will be covered in the homework section.

6.3 Design

Symbolic execution runtime manages condition branches where symbolic variables are used. Speficially it manages "state" which values satisfies or not each branch conditions.

6.3.1 State

```c
// symbolic-example.c
#include <stdio.h>

extern void __make_symbolic();

```

```
6  int main() {
7      int i,j;
8      __make_symbolic(sizeof(int), &i);
9      __make_symbolic(sizeof(int), &j);
10     printf("%d, %d\n", i,j);
11     // state 0
12     if (i == 0) { // state 1
13         if (j == 123214125) { // state 2
14             printf("S1\n");
15         } else { // state 3
16             printf("S2\n");
17         }
18     } else { // state4
19         if (j != 88148128) { // state 5
20             printf("S3\n");
21         } else { // state 6
22             printf("S4\n");
23             return 123;
24         }
25     }
26 }
```

In our example, there are a total of six states. A new state is created whenever a branch condition incorporates a symbolic value. Let's trace this process starting from line 7.

Lines 7 declares two integer variables, `i` and `j`. Lines 8 and 9 indicate that `i` and `j` are symbolic variables, meaning their concrete values will be supplied by the symbolic execution runtime.

At line 12, the symbolic variable `i` is compared. At this point, a new state, State 1, is created. Additionally, another state, State 2 (line 18), is also generated. When a new state is required due to a branch, two states are created through a process called ***forking***.

A fork involves copying the current state and only changing its predicate by negating it. For example, when creating State 2 and State 3, they will have identical content except for their predicates. State 2 will have the "equal" predicate, while State 3 will have the "not equal" predicate.

At line 13, the symbolic variable `j` is used in a branch condition, leading to the creation of State 3 and State 4 (line 15). Finally, line 19 creates State 5 and State 6 (line 22). The figure below illustrates each of these state nodes.

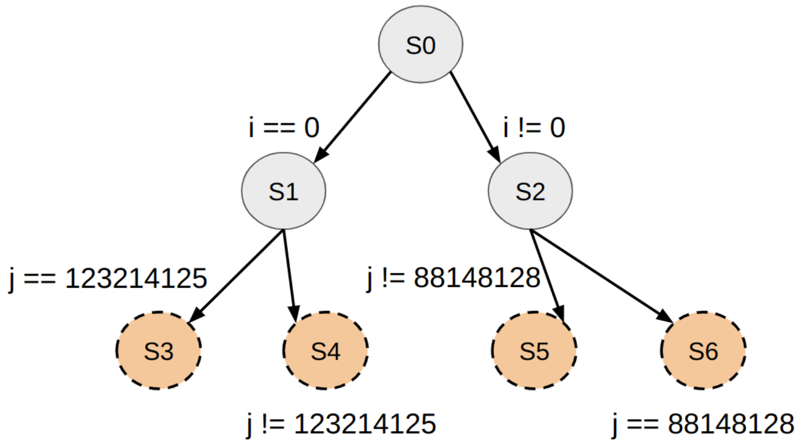


Figure 6.1: Symbolic State

This gives us a total of six states, excluding the initial state, ‘S0’.

6.3.2 Static Analysis vs. Dynamic Management

I believe there are two main approaches to managing states: static analysis or dynamic analysis.

If you delegate state creation to runtime, dynamic analysis comes into play. When the target program starts, nothing special happens initially. However, when a branch involving a symbolic variable is encountered, the system detects the relevant instruction (like `icmp`) and dynamically creates a new state. This approach is typically used when the target program is assumed to be, or limited to, a binary.

When source code is available, we can opt for static analysis. This allows us to identify how many branch conditions involve symbolic variables at compile time (during instrumentation). This way, potential states can be calculated upfront. However, static analysis can’t pre-calculate all possible states. This challenge can be mitigated with concolic testing [25] (combining concrete and symbolic execution); by executing concrete inputs and using the feedback, it can prune useless states. We’ll detail this in the homework section.

In this book, we’re opting for a static analysis approach to create all explicit states (as demonstrated in the limited example above). For instance, our implementation achieves 100% coverage on the working example. However, if a conditional branch requires an unknown state—a combination of existing states that can’t be resolved during static analysis—it introduces the problems of state pruning and state explosion. We’ll delve into these challenges further in the homework section.

Returning to our specific scope, we instrument the encoding for all explicit states. Our runtime then resolves the resulting equations using Z3 [26], a powerful theorem prover.

6.3.3 Interface (Harness)

In our example, the `__make_symbolic()` functions at lines 8 and 9 serve as the interface between the target program and the symbolic runtime. During each run, the symbolic runtime uses Z3 to supply resolved values for these variables. Therefore, when performing symbolic testing on any program, you'll need to select your symbolic variables and initialize them using this function, essentially creating a testing harness.

6.4 Implementation

As always, our implementation consists of two parts: the instrumentation module and the runtime.

6.4.1 Instrumentation

Our instrumentation strategy is divided into four steps:

- **Collecting Symbolic Variables**: We capture symbolic variables by identifying calls to the `__make_symbolic()` initialization function.
- **State Creation**: By iterating through basic blocks, we look for `icmp` instructions. If a symbolic variable is involved, we create a new state by forking
- **State Initialization via Runtime**: Each state node is encoded and passed to a runtime initialization function. Only the leaf state nodes are serialized and passed into this function.
- **Instrumentation**: The serialized states are instrumented within a constructor function, and all constraints from the target program are initialized in the runtime

Let's begin by looking at the data structure.

```rust
// instrument/src/symbolic.rs

pub const VAR_KIND: i8 = 0;
pub const CONST_KIND: i8 = 1;

pub const PREDICATE_EQ: i8 = 0;
pub const PREDICATE_NE: i8 = 1;
pub const PREDICATE_SLT: i8 = 2;
pub const PREDICATE_SLE: i8 = 3;
pub const PREDICATE_SGT: i8 = 4;
pub const PREDICATE_SGE: i8 = 5;

#[derive(Debug, Clone)]
enum Operand<'ctx> {
    Var(PointerValue<'ctx>),
    Const(i64),
}
impl<'ctx> Operand<'ctx> {
    fn is_const(&self) -> bool {
        if let Self::Const(_) = self {
            return true;
        } else {
            return false;
```

```rust
        }
    }
}

#[derive(Debug, Clone)]
struct Constraint<'ctx> {
    left_operand: Operand<'ctx>,
    right_operand: Operand<'ctx>,
    predicate: IntPredicate,
}

impl<'ctx> Constraint<'ctx> {
    fn new(
        left_operand: Operand<'ctx>,
        right_operand: Operand<'ctx>,
        predicate: IntPredicate,
    ) -> Self {
        Self {
            left_operand,
            right_operand,
            predicate,
        }
    }

    fn predicate_to_i8(&self) -> i8 {
        match &self.predicate {
            IntPredicate::EQ => PREDICATE_EQ,
            IntPredicate::NE => PREDICATE_NE,
            IntPredicate::SLT => PREDICATE_SLT,
            IntPredicate::SLE => PREDICATE_SLE,
            IntPredicate::SGT => PREDICATE_SGT,
            IntPredicate::SGE => PREDICATE_SGE,
            _ => unreachable!("unsigned operation is not supported"),
        }
    }

    fn negate(&self) -> Self {
        let predicate = match self.predicate {
            IntPredicate::EQ => IntPredicate::NE,
            IntPredicate::NE => IntPredicate::EQ,
```

```rust
            IntPredicate::SLT => IntPredicate::SGE,
            IntPredicate::SLE => IntPredicate::SGT,
            IntPredicate::SGT => IntPredicate::SLE,
            IntPredicate::SGE => IntPredicate::SLT,
            _ => unreachable!("unsigned operation is not supported"),
        };
        Self::new(
            self.left_operand.clone(),
            self.right_operand.clone(),
            predicate,
        )
    }
}
```

The `Operand` struct represent whether the operand is symbolic variable or constant value. In the working example, the symbolic variables `i` and `j` correspond to `Var` and constant values correspond to `Const` type such as `123214125` and `88148128`.

The `Constrant` struct represents a branch condition. Branch condition contains left and right operand and predicator. For example, `icmp eq i32 %1, 555` includes left operand `%1`(`Var`), right operand(`555`(`Const`)) and predicator (`IntPredicate::EQ`). We consider only six type predicator, ==, `!=`, <, <=, >, and >=. You can include other predicators for extension.

The function `predicate_to_i8()` is used when a state is serialized. The `negate()` function

The `negate()` function negate the predicator and other properties are remain.

The `Operand` struct tells us if an operand is a symbolic variable or a constant value. In our example, symbolic variables like `i` and `j` would be represented by the `Var` type, while constant values such as 123214125 and 88148128 would be `Const` types.

The `Constraint` struct represents a branch condition. It holds a left operand, a right operand, and a predicate. For instance, an instruction like `icmp eq i32 %1, 555` would have `%1` (`Var`) as its left operand, 555 (`Const`) as its right operand, and `IntPredicate::EQ` as its predicate. We're focusing on six types of predicates: `==`, `!=`, <, <=, >, and >=. You can extend this to include others if needed.

The `predicate_to_i8()` function is used when a state is serialized.

The `negate()` function, on the other hand, simply negates the predicate while keeping all other properties unchanged.

Now we introduce state structure:

```rust
// instrument/src/symbolic.rs

#[derive(Debug, Clone)]
struct State<'ctx> {
    id: i64,
    path_constraints: Vec<Constraint<'ctx>>,
    is_leaf: bool,
```

```
}

impl<'ctx> State<'ctx> {
    fn new() -> Self {
        Self {
            id: 0,
            path_constraints: vec![],
            is_leaf: false,
        }
    }

    fn fork(&self, constraint: Constraint<'ctx>) -> (Self, Self) {
        let mut id = CONSTRAINT_ID.lock().unwrap();
        let (tid, fid) = (*id + 1, *id + 2);
        *id += 2;
        let (mut copied_state_t, mut copied_state_f) = (self.clone(), self.clone());
        copied_state_t.path_constraints.push(constraint.clone());
        copied_state_f.path_constraints.push(constraint.negate());
        copied_state_t.id = tid;
        copied_state_f.id = fid;
        (copied_state_t, copied_state_f)
    }
}
```

Each state is assigned a unique ID, which serves as its identifier. The path_constraints field holds a set of branch conditions. It's crucial that a branch condition within a nested structure preserves the constraints from its previous state. Let's look at an example to understand why preservation is required:

```c
1 if (a == 1) {
2   if (b == 1) {
3     ...
4   }
5 }
```

To reach line 3, we must satisfy two conditions: `a == 1` and `b == 1`. If either of these isn't met, line 3 will never execute. This highlights why we need to be aware of the current state before forking a new one. In the example above, at line 2, we have one constraint: `a == 1`. When we create a new state, we append the new branch condition to the current state, resulting in `a == 1 && b == 1`. This is precisely why `path_constraints` is implemented as a list (or vector) type.

To create a fork, we generate two new IDs and clone the current state twice. One clone represents the true branch, while the other represents the false branch. For the false branch, we simply negate the predicate. The implementation is straightforward; you can examine the `fork()` function for details.

Now, let's dive into the main logic of our instrumentation. We'll focus on the core components here; you can find the full implementation in the repository.

```rust
// instrument/src/symbolic.rs

#[derive(Default)]
pub struct SymbolicModule {}

fn build_sym_ptrs<'ctx>(
    context: &'ctx Context,
    module: &Module<'ctx>,
    builder: &Builder<'ctx>,
) -> Result<Vec<PointerValue<'ctx>>> {
    let mut symbolic_ptr_inits = HashSet::new();
    let funcs: Vec<_> = module.get_functions().collect();
    for func in funcs {
        if can_skip_instrument(&func) {
            continue;
        }
        for basic_blk in func.get_basic_blocks() {
            for instr in basic_blk.get_instructions() {
                if instr.get_opcode() == InstructionOpcode::Call && instr.get_num_operands() == 3 {
                    let func_ptr = instr.get_operand(2).unwrap().left().unwrap();
                    let func_str = cstr_to_str(func_ptr.get_name());
                    if func_str == SYMBOLIC_MAKE_VAR { // __make_symbolic()
                        let var_ptr = instr
                            .get_operand(1)
                            .unwrap()
                            .left()
                            .unwrap()
                            .into_pointer_value();
                        symbolic_ptr_inits.insert(var_ptr);

                        // install <address, pointer value> mapping
                        builder.position_before(&instr);
                        build_sym_make_prep(context, module, builder, var_ptr)?;
                    }
```

```
                }
            }
        }
    }
    Ok(symbolic_ptr_inits.iter().cloned().collect())
}
```

The function `build_sym_ptr()` collects all symbolic variables by looking the function `__make_symbolic()`.

```rust
// instrument/src/symbolic.rs

fn get_sym_ptr_operand<'ctx>(
    sym_ptrs: &Vec<PointerValue<'ctx>>,
    op: BasicValueEnum<'ctx>,
) -> Option<Operand<'ctx>> {
    if op.is_int_value() {
        let op = op.into_int_value();
        if op.is_const() {
            return
                Some(Operand::Const(op.get_sign_extended_constant().unwrap()));
        } else {
            let mut instr = op.as_instruction().unwrap();
            loop {
                match instr.get_opcode() {
                    InstructionOpcode::Load => {
                        let ptr = instr.get_operand(0).
                                                     unwrap().
                                                     left().unwrap();
                        for sym_ptr in sym_ptrs {
                            if *sym_ptr == ptr {
                                return Some(Operand::Var(*sym_ptr));
                            }
                        }
                        return None;
                    }
                    _ => {
                        if let Some(prev_instr) = instr.get_previous_instruction() {
                            instr = prev_instr;
                        } else {
                            return None;
```

```
                        }
                    }
                }
            }
        }
    }
    None
}
```

The `get_sym_ptr_operand()` function is designed to retrieve symbolic variables within branch conditions.

- If a branch condition's operand is a constant integer, the function returns the constant, wrapped appropriately.
- If the operand is a symbolic variable, it returns the symbolic variable, also wrapped.
- If the operand is neither (i.e., not a symbolic variable), None is returned, indicating that no symbolic variable is involved in that particular branch condition.

The code below illustrates the state initialization instrumentation. We split into two parts because of the page limit.

```rust
// [1-1]
// instrument/src/symbolic.rs

impl InstrumentModule for SymbolicModule {
    fn instrument<'ctx>(
        &self,
        context: &'ctx Context,
        module: &Module<'ctx>,
        builder: &Builder<'ctx>,
    ) -> Result<()> {
        let sym_ptrs = build_sym_ptrs(context, module, builder)?;
        let mut serialized = vec![];
        let funcs: Vec<_> = module.get_functions().collect();
        for func in funcs {
            let mut states: HashMap<i64, State> = HashMap::new();
            let mut bb_id = HashMap::new();
            if let Some(first_bb) = func.get_first_basic_block() {
                let first_bb_addr = first_bb.as_mut_ptr() as usize;
                bb_id.insert(first_bb_addr, 0);
                states.insert(0, State::new());
            }
            for basic_blk in func.get_basic_blocks() {
                for instr in basic_blk.get_instructions() {
                    .... // refer the code snippet below [1-2]
```

```
25         }
26         instrumented_blks.insert(basic_blk);
27       }
28       if states.len() > 1 {
29         serialized.push(serailize_constraints(&states));
30       }
31     }
32     if !serialized.is_empty() {
33       let constructor = build_symbolic_init(context, module, builder, serialized)?;
34       build_ctros(context, module, constructor)?;
35     }
36
37     // Verify instrumented IRs
38     module_verify(module)
39   }
40 }
```

Our goal is to instrument serialized constraints so the runtime can recognize which constraints exist.

We begin by identifying the symbolic variables. We insert State 0 at the first basic block of each function, meaning our current implementation is limited to intra-procedural analysis. State 0 doesn't contain any constraints itself.

The core logic resides in the loop from lines 22 to 27, with detailed code and descriptions provided below.

Once states are created, they're serialized and added to the `serialized` vector. After iterating through all states, these serialized constraints are then instrumented within a constructor function.

```rust
// instrument/src/symbolic.rs
// [1-2]

match instr.get_opcode() {
 InstructionOpcode::Br => {
 // if the current basic block has a state and the terminator is not conditional, we mark them as leaf
  if !instr.is_conditional() {
   let bb_addr = basic_blk.as_mut_ptr() as usize;
   if let Some(id) = bb_id.get(&bb_addr) {
    if let Some(state) = states.get_mut(&id) {
     state.is_leaf = true;
    }
   }
  }
```

```
}
InstructionOpcode::ICmp => {
  if let Some(br_instr) = instr.get_next_instruction() {
    if br_instr.get_opcode() == InstructionOpcode::Br && br_instr.is_conditional() {
      let (left_op, right_op) = (
        instr.get_operand(0).unwrap().left().unwrap(),
        instr.get_operand(1).unwrap().left().unwrap(),
      );
      let (tbr, fbr) = (
        br_instr.get_operand(2).unwrap().right().unwrap(),
        br_instr.get_operand(1).unwrap().right().unwrap(),
      );
      match (
        get_sym_ptr_operand(&sym_ptrs, left_op),
        get_sym_ptr_operand(&sym_ptrs, right_op),
      ) {
        (Some(left_operand), Some(right_operand)) => {
        // ignore if both operands are constant
          if left_operand.is_const() && right_operand.is_const() {
            continue;
          }
          let predicate = instr.get_icmp_predicate().unwrap();
          let constraint = Constraint::new(
            left_operand,
            right_operand,
            predicate,
          );
          let bb_addr = basic_blk.as_mut_ptr() as usize;
          if let Some(id) = bb_id.get(&bb_addr) {
            if let Some(state) = states.get(&id) {
              let (state_t, state_f) = state.fork(constraint);
                bb_id.insert(
                  tbr.as_mut_ptr() as usize,
                  state_t.id,
                );
                bb_id.insert(
                  fbr.as_mut_ptr() as usize,
                  state_f.id,
                );
                states.insert(state_t.id, state_t);
```

```
                  states.insert(state_f.id, state_f);
              }
            } else {
              let state = states.get(&0).unwrap();
              let (state_t, state_f) = state.fork(constraint.clone());
              bb_id.insert(tbr.as_mut_ptr() as usize, state_t.id);
              bb_id.insert(fbr.as_mut_ptr() as usize, state_f.id);
              states.insert(state_t.id, state_t);
              states.insert(state_f.id, state_f);
            }
          }
          _ => {}
        }
      }
    }
  }
  _ => {}
}
```

The code above represents the main handler for collecting states. It identifies conditional branch instructions and then determines if their operands correspond to symbolic variables. If a symbolic variable is found, it creates a `Constraint` struct and generates two new states by forking. As we mentioned earlier, to correctly handle nested branches, the previous state is queried using its basic block address and state ID.

When a new state is added, the addresses for both the true and false branches are stored, along with the IDs of the two newly generated states. This ensures that the current state information propagates to future states, allowing them to correctly determine which states need to be considered.

The following function serializes the set of constraints.

```rust
// instrument/src/symbolic.rs

#[derive(Debug)]
pub struct ConstraintSerialized {
    pub id: i64,
    pub left_operand_kind: i8,
    pub left_operand_val: i64,
    pub right_operand_kind: i8,
    pub right_operand_val: i64,
    pub predicate: i8,
}

impl ConstraintSerialized {
```

```
    pub fn new(
        id: i64,
        left_operand_kind: i8,
        left_operand_val: i64,
        right_operand_kind: i8,
        right_operand_val: i64,
        predicate: i8,
    ) -> Self {
        Self {
            id,
            left_operand_kind,
            left_operand_val,
            right_operand_kind,
            right_operand_val,
            predicate,
        }
    }
}

fn serailize_constraints<'ctx>(states: &HashMap<i64, State>) ->
Vec<ConstraintSerialized> {
    let mut constraints = vec![];
    for (_, state) in states {
        if state.is_leaf {
            for constraint in &state.path_constraints {
                let (left_kind, left_op) = match
                constraint.left_operand {
                    Operand::Var(var) => (VAR_KIND,
                    var.as_value_ref() as i64),
                    Operand::Const(v) => (CONST_KIND, v),
                };
                let (right_kind, right_op) = match
                constraint.right_operand {
                    Operand::Var(var) => (VAR_KIND,
                    var.as_value_ref() as i64),
                    Operand::Const(v) => (CONST_KIND, v),
                };
                let predicate = constraint.predicate_to_i8();
                constraints.push(ConstraintSerialized::new(
                    state.id, left_kind, left_op, right_kind,
                    right_op, predicate,
                ));
            }
```

```
            }
        }
        constraints
    }
```

The only leaf nodes are instrumented as follows in the given working example.

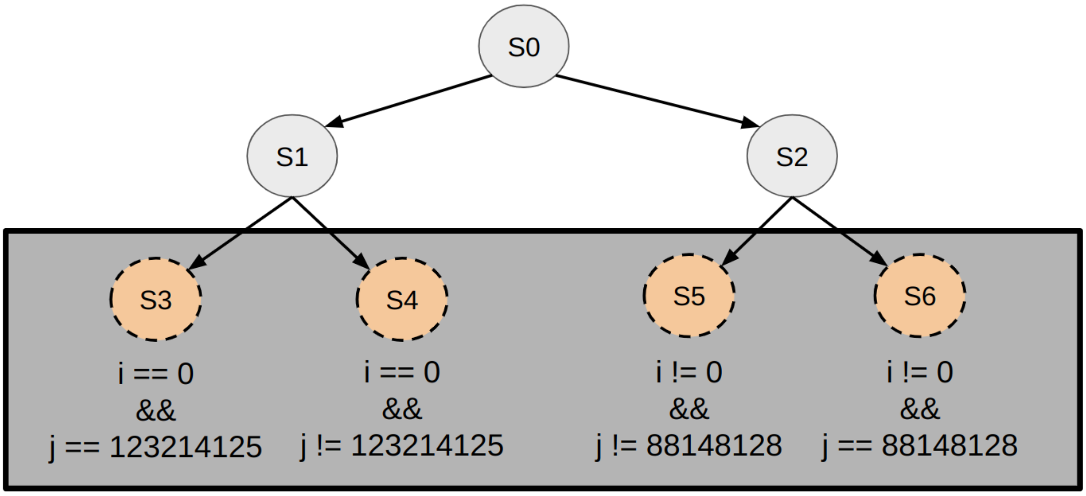


Figure 6.2: Leaf State Nodes

Finally, each serialized states are instrumented as follows:

```
define i32 @main() #0 !dbg !27 {
  ...
  call void @__symbolic_make_prepare(ptr %3, i64 106096555351408), !dbg !36
  call void (i64, ptr, ...) @__make_symbolic(i64 noundef 4, ptr noundef %3), !dbg !36
  call void @__symbolic_make_prepare(ptr %4, i64 106096555351136), !dbg !37
  call void (i64, ptr, ...) @__make_symbolic(i64 noundef 4, ptr noundef %4), !dbg !37
  ...
  ...

declare void @__symbolic_module_add_sym(i64, i8, i64, i8, i64, i8)

define void @__symbolic_init(i64 %0, i8 %1, i64 %2, i8 %3, i64 %4, i8 %5) {
__symbolic_init_entry:
```

LLVM-IR

```
    call void @__symbolic_module_add_sym(i64 5, i8 0, i64
    106096555351408, i8 1, i64 0, i8 1)
    call void @__symbolic_module_add_sym(i64 5, i8 0, i64
    106096555351136, i8 1, i64 88148128, i8 1)
    call void @__symbolic_module_add_sym(i64 4, i8 0, i64
    106096555351408, i8 1, i64 0, i8 0)
    call void @__symbolic_module_add_sym(i64 4, i8 0, i64
    106096555351136, i8 1, i64 123214125, i8 1)
    call void @__symbolic_module_add_sym(i64 6, i8 0, i64
    106096555351408, i8 1, i64 0, i8 1)
    call void @__symbolic_module_add_sym(i64 6, i8 0, i64
    106096555351136, i8 1, i64 88148128, i8 0)
    call void @__symbolic_module_add_sym(i64 3, i8 0, i64
    106096555351408, i8 1, i64 0, i8 0)
    call void @__symbolic_module_add_sym(i64 3, i8 0, i64
    106096555351136, i8 1, i64 123214125, i8 0)
    ret void
}
```

Since only the leaf nodes are instrumented, there are a total of eight states, resulting from the forking of four distinct states.

6.4.2 Runtime

The core of our runtime implementation involves interacting with the Z3 solver.

Our first task is to add constraints for each state ID.

```rust
symbolic_runtime/src/runtime.rs

#[no_mangle]
pub extern "C" fn __symbolic_module_add_sym(
    id: i64,
    left_operand_kind: i8,
    left_operand_val: i64,
    right_operand_kind: i8,
    right_operand_val: i64,
    predicate: i8,
) {
    CONSTRAINTS.write().unwrap().add_constraint(
        id,
        left_operand_kind,
        left_operand_val,
        right_operand_kind,
        right_operand_val,
        predicate,
    );
```

```
}
```

The global singleton `CONSTRAINTS` holds all constraints associated with each state ID. This means that all constraints within a given ID are considered collectively to find a solution.

```rust
// symbolic_runtime/src/symbolic.rs

lazy_static::lazy_static! {
    pub static ref CONSTRAINTS: RwLock<Solver> =
    RwLock::new(Solver::new());
}

pub struct Solver {
    pub constraints: HashMap<i64, Vec<ConstraintSerialized>>,
}

impl Solver {
    fn new() -> Self {
        Self {
            constraints: HashMap::new(),
        }
    }

    pub fn add_constraint(
        &mut self,
        id: i64,
        left_operand_kind: i8,
        left_operand_val: i64,
        right_operand_kind: i8,
        right_operand_val: i64,
        predicate: i8,
    ) {
        self.constraints
            .entry(id)
            .or_insert(vec![])
            .push(ConstraintSerialized::new(
                id,
                left_operand_kind,
                left_operand_val,
                right_operand_kind,
                right_operand_val,
                predicate,
```

```
            ));
    }
    ...
    ...
}
```

The next exposed runtime function call is `__symbolic_make_prepare()`, which adds a symbolic variable identifier with a tuple <pointer address, identifier>.

```rust
// symbolic_runtime/src/runtime.rs

#[no_mangle]
pub extern "C" fn __symbolic_make_prepare(ptr: *mut libc::c_void, addr: i64) {
    ADDRS.write().unwrap().insert(ptr as i32, addr);
}
```

The next exposed runtime function call is `__make_symbolic()`, which supplies solutions from the solver.

```rust
// symbolic_runtime/src/runtime.rs

#[no_mangle]
pub extern "C" fn __make_symbolic(typ_size: usize, ptr: *mut libc::c_void) {
    if !ptr.is_null() {
        let id = select_id();
        if id.is_none() {
            return;
        }
        let addr = {
            let addrs = ADDRS.read().unwrap();
            *addrs.get(&(ptr as i32)).unwrap()
        };

        let id = id.unwrap();
        if let Some(solutions) = CONSTRAINTS.read().unwrap().solve(id) {
            for (sym_addr, solution) in solutions {
                if sym_addr == addr {
                    match typ_size {
                        1 => unsafe {
                            std::ptr::write(ptr as *mut libc::c_char, solution as i8);
                        },
```

```
                        4 => unsafe {
                            std::ptr::write(ptr as *mut libc::c_int, solution as i32);
                        },
                        _ => unsafe {
                            std::ptr::write(ptr as *mut libc::c_long, solution);
                        },
                    }
                }
            }
        }
    }
}
```

It first randomly selects a state ID and retrieves the symbolic variable identifier associated with the given pointer address. This ensures the solution is supplied to the exact symbolic variable.

If a solution is found for the given constraint set, it is written into the symbolic variable's pointer. Now, let's examine how to interact with the Z3 solver to find a solution based on these constraints.

```rust
// symbolic_runtime/src/symbolic.rs

fn get_symbolic_val<'ctx>(
    ctx: &'ctx Context,
    var_names: &mut HashMap<i64, String>,
    var_cnt: &mut u64,
    kind: i8,
    val: i64,
) -> Int<'ctx> {
    if kind == VAR_KIND {
        if var_names.get(&val).is_none() {
            let var_name = format!("var{}", var_cnt);
            *var_cnt += 1;
            var_names.insert(val, var_name);
        }
        let var_name = var_names.get(&val).unwrap().clone();
        Int::new_const(&ctx, var_name.clone())
    } else {
        Int::from_i64(&ctx, val)
    }
}

```

```
fn set_solver_timeout(ctx: &Context, solver: &Z3Solver, ms: u32) {
    let mut params = Params::new(&ctx);
    params.set_u32("timeout", ms);
    solver.set_params(&params);
}

pub fn select_id() -> Option<i64> {
    let constraints = CONSTRAINTS.read().unwrap();
    let ids: Vec<_> = constraints.constraints.keys().collect();
    if !ids.is_empty() {
        Some(**ids.choose(&mut rand::rng()).unwrap())
    } else {
        None
    }
}

pub fn solve(&self, id: i64) -> Option<Vec<(i64, i64)>> {
    let cfg = Config::new();
    let ctx = Context::new(&cfg);
    let solver = Z3Solver::new(&ctx);
    let mut stmts = vec![];
    let mut syms = HashMap::new();
    let mut var_names = HashMap::new();
    let mut var_cnt = 0;
    let constraints = self.constraints.get(&id).unwrap();
    for constraint in constraints {
        let (left_sym_val, right_sym_val) = (
            get_symbolic_val(
                &ctx,
                &mut var_names,
                &mut var_cnt,
                constraint.left_operand_kind,
                constraint.left_operand_val,
            ),
            get_symbolic_val(
                &ctx,
                &mut var_names,
                &mut var_cnt,
                constraint.right_operand_kind,
                constraint.right_operand_val,
            ),
```

```
        );
        let stmt;
        match constraint.predicate {
            PREDICATE_EQ => {
                stmt = left_sym_val._eq(&right_sym_val);
            }
            PREDICATE_NE => {
                stmt = left_sym_val._eq(&right_sym_val).not();
            }
            PREDICATE_SLT => {
                stmt = left_sym_val.lt(&right_sym_val);
            }
            PREDICATE_SLE => {
                stmt = left_sym_val.le(&right_sym_val);
            }
            PREDICATE_SGT => {
                stmt = left_sym_val.gt(&right_sym_val);
            }
            PREDICATE_SGE => {
                stmt = left_sym_val.ge(&right_sym_val);
            }
            _ => unreachable!("unexpected predicate: {}", constraint.predicate),
        };
        syms.insert(constraint.left_operand_val, left_sym_val);
        syms.insert(constraint.right_operand_val, right_sym_val);
        stmts.push(stmt.clone());
        solver.assert(&stmt);
    }
    let combined = Bool::and(&ctx, &stmts.iter().collect::<Vec<_>>());
    set_solver_timeout(&ctx, &solver, 5000);
    match solver.check() {
        SatResult::Sat => {
            let model = solver.get_model().unwrap();
            let mut solutions = vec![];
            for (val, sym) in &syms {
                let solution = model.eval(sym).unwrap().as_i64().unwrap();
                if var_names.get(val).is_some() {
                    solutions.push((*val, solution));
                }
```

```
                }
                Some(solutions)
            }
            SatResult::Unsat => {
                println!("UNSAT: {:?}", combined);
                None
            }
            SatResult::Unknown => {
                println!("UNKNOWN: {:?}", combined);
                None
            }
        }
    }
```

The `get_symbolic_val()` function is a utility that generates temporary symbolic variable names for use with the solver.

`set_solver_timeout()` adds a time constraint to the solver; in our implementation, this timeout is hardcoded to 5 seconds.

`select_id()` simply chooses a random state ID.

The `solve()` function contains the core logic for building statements and querying the solver.

Before we deep dive into the `solve()` body, let's first explore how to use Z3.

```smt
// example-code/symbolic/example.smt

; Declare integer variables x and y.
(declare-const x Int)
(declare-const y Int)

; Add constraints (assertions) to the solver.
; x must be greater than or equal to 0.
(assert (>= x 0))
; y must be greater than or equal to 0.
(assert (>= y 0))
; The sum of x and y must be 10.
(assert (= (+ x y) 10))
; x must be less than y.
(assert (< x y))

; Check if the current set of assertions is satisfiable.
(check-sat)

; If satisfiable, retrieve and print the model (variable assignments).
```

```
21 (get-model)
```

This example code illustrates how to construct a statement and query the solver.

- At lines 4-5, we declare two symbolic variables, named “x” and “y”.
- From lines 7-15, various constraints are added.
- At line 18, we check if a solution exists for the given constraints.
- Finally, at line 20, if the constraints are satisfiable, we retrieve a concrete solution.

Here is the result:

```Shell
❯ z3 example.smt
sat
(
  (define-fun y () Int
    6)
  (define-fun x () Int
    4)
)
```

The `solve()` function encodes constraints using the Rust Z3 binding.

First, from lines 50 to 64, we create symbolic variable names. Then, from lines 66 to 86, the left and right symbolic variables are constructed into statements based on the given predicate. At line 90, each built statement is recorded. This process repeats for every constraint.

At line 92, all these statements are joined using the `AND` (&&) operator. Line 94 checks if the combined statement is satisfiable. If it is, the solutions for each symbolic variable are collected. Finally, these solutions are written back into the program via the `__make_symbolic()` function.

We have implemented a very basic symbolic execution engine, and there is significant room for improvement to make it more practical.

6.4.2.1 Homework

Exercise 6.1 We suggest the following topics for further study and exploration:

Topic #1: Extending Symbolic Variable Type: Our current implementation supports symbolic variable types only for integers and characters. Readers may extend this to broader types, such as arrays.

Topic #2: Concolic Execution: Our current implementation is effective only for simple examples, like the working example, where all states can be identified easily without any loss. In the following example, we will introduce a genuinely difficult problem and explain why our current implementation cannot solve it.

```C
#define H 7
#define W 11
char maze[H][W] = { "+-+---+---+",
                    "|S|     |#|",
                    "| | --+ | |",
```

```
6                       "| |   | | |",
7                       "| +-- | | |",
8                       "|     |   |",
9                       "+-----+---+" };
10 char* arr[28];
11 int x = 1; int y = 1; int i = 0;
12 #define ITERS 28
13 __make_symbolic(ITERS, arr);
14 while (i < ITERS) {
15   switch (arr[i]) {
16     case 'w': y--; break;
17     case 's': y++; break;
18     case 'a': x--; break;
19     case 'd': x++; break;
20     default: printf("Wrong command\n"); exit(-1);
21   }
22   if (maze[y][x] == '#') {
23      printf ("You win!\n"); exit (1);
24   }
25   if (maze[y][x] != ' ' && !((y == 2 && maze[y][x] == '|' && x > 0 && x < W))) {
26      x = ox;
27    y = oy;
28   }
29   if (ox==x && oy==y){
30     printf("You lose\n");
31     exit(-2);
32   }
33 }
```

The example above is a maze program, abstracted from the [27] problem.

The `arr` variable holds a sequence of arrow commands. Multiple solutions exist, such as `ssssddddwwaawwddddssssddwwww`. Our current symbolic execution can identify four possible states within the switch statement. However, the core challenge lies at line 22. While we can generate 'w', 's', 'a', or 'd' for each state, the condition `maze[y][x] == '#'` is not directly tied to these four explicit states. This means the program's implicit states (like the maze layout) need to be explored by generating various sequences of arrow commands.

Consider whether this could be solved with random generation. Our current implementation can generate the four appropriate arrows ('w', 's', 'a', 'd'), but it doesn't know which arrow should be placed at which element within the sequence. If we were to generate these randomly, the probability of reaching line 23 would be roughly `0.25 ** 28 =1.3877787807814457e-17` —an utterly impractical solution in the real world.

Here, the concept of ***state explosion*** becomes apparent. Randomly mixing states often fails to satisfy complex branch conditions. To mitigate state explosion, ***state pruning*** can be adopted, where useless states are discarded. For example, a command like "ssssssss" will not be a maze solution, and the program will likely hit line 30 at the fifth 's'. At that point, having covered the line 30 branch, we can prioritize other uncovered branches. This suggests using backtracking to quickly revert to a state before a previous coverage point was found. We would then go back to the command "ssss" and try generating 'w', 'a', or 'd' for the fifth character. This would yield three new commands: "sssw", "sssa", or "sssd". If "sssw" and "sssd" hit line 30 again, we would prune them from the set of states and continuously proceed with the remaining viable commands, repeating this process.

For this homework, it's sufficient to understand this problem. Passionate readers may attempt to implement a symbolic execution engine capable of solving the maze problem within a reasonable time budget. Implementing state pruning and scheduling techniques would be necessary to reduce useless exploration and save time. Some readers might realize at this point that the basis of state pruning often relies on coverage. If a state has been explored sufficiently, and it's confirmed that it won't lead to further new coverage, that state can be safely removed to explore uncovered code quickly.

Part VI

Delta Debugging

“Buggy input found. Is it the minimized one?”

```
Input1: ‘0x1aa2bbac1addabb8b1b2” ✗
Input2: ‘0x1aa2bbac1add”         ✗
Input3: ‘0x1aa2”                 ✓
Input4: ‘0x1aa2bba”              ✗
```

7. Introduction to Delta Debugging

7.1 Introduction

Let’s assume our fuzzer has found a crash with an input 100 bytes long. A programmer can reproduce and confirm this bug using that specific seed. However, this isn’t the final step. While we’ve identified a crash-inducing input, is it truly minimal? There might be another input that causes the same crash but is significantly shorter. A minimized input greatly aids the programmer in understanding why this input led to a bug. Simply put, when comparing a 3-byte bug-inducing input with a 100-byte one, which is easier to debug? Absolutely the former.

This brings us to Delta Debugging (DD) [28], a methodology for finding minimal failure-inducing inputs. The ultimate goal of this chapter is to integrate DD into our fuzzer, enabling it to automatically reduce crash-inducing seeds.

7.2 Simple Idea (Binary Search)

Imagine our target input contains the characters ‘1’ and ‘7’. A crashing input might be “12345678” because it includes both.

Perhaps we can apply a binary search approach to find the minimal failure-inducing input. Let’s explore that.

Step	Test	Result
0	**1 2 3 4 5 6 7 8**	
1	**1 2 3 4**	Pass
2	. . . . **5 6 7 8**	Pass
3	**1 2**	Pass
4	. . **3 4**	Pass
5	. . . . **5 6** . .	Pass
6	. . . . . . **7 8**	Pass
7	**1**	Pass
8	. **2**	Pass

Step	Test	Result
9	. . **3**	Pass
10	. . . **4**	Pass
11	. . . . **5** . . .	Pass
12	. . . . . **6** . .	Pass
13	. . . . . . **7** .	Pass
14	. . . . **8**	Pass

Even with a failure-inducing input, binary search failed to find the bug. This is because the process doesn't account for the **"complement"** The complement is the combined set of elements excluding the current delta. For instance, in Step 3, the complement of the input ("12") is "345678" Thus, formally, the complement is defined as:

$$c_x = \text{failure-inducing input (e.g., 12345678 in the example)} \tag{7.1}$$

$$\Delta_i = \text{delta of } \ c_x \tag{7.2}$$

$$\nabla_i(\text{Complement}) = c_x - \Delta_i \tag{7.3}$$

By definition, in Step 3, the complement is "12345678" - "12" = "345678". Therefore, considering the complement is key to DD, and we will now introduce the **1-minimal algorithm**.

7.3 Local Minmum and Global Minimum

Finding a global minimum is a time-consuming and challenging problem. In the real world, finding a local minimum is usually sufficient.

- **n-minimal**: A failure-inducing input where the bug disappears if any n elements are removed.
- **1-minimal**: A failure-inducing input where the bug disappears if any single element is removed.

Finding an n-minimal input is more challenging than finding a 1-minimal input because the 1-minimal algorithm is greedy; it stops as soon as the bug disappears.

In the previous example, with the input "12345678", both n-minimal and 1-minimal algorithms successfully find the global minimum, which is "17". Now, let's explore a more complex buggy condition.

Assume the buggy condition is as follows:

- If the input contains both 'A' and 'B', a bug is triggered.
- If the input contains both 'A' and 'C', a bug is triggered.

Given the input "ABCDE", let's assume "ABC" is an intermediate result of the 1-minimal algorithm. In the next step, when it attempts to delete 'A', the resulting "BC" does not report a bug because it satisfies neither condition 1 nor condition 2. If either 'B' or 'C' were removed, the results ('AC' or 'AB') would trigger a bug. However, since the algorithm found a non-triggering condition ("BC"), the 1-minimal algorithm will incorrectly conclude that the minimized result is "ABC". The correct answers, however, are "AB" or "AC". Therefore, the 1-minimal algorithm does not guarantee a global minimum.

7.4 1-minimal Algorithm

We'll now introduce the 1-minimal algorithm from the paper [29].

The high-level idea is straightforward: the algorithm repeatedly divides the input into a **delta (subset)** and its **complement**, then runs a test (or oracle) to see if a bug appears. This process continues until no bug is found or the granularity can't be refined further.

We attach the intuitive algorithm here from the paper, so called, **"ddmin"**.

Minimizing Delta Debugging Algorithm

Let *test* and $c_{\text{✘}}$ be given such that $test(\emptyset) = \text{✔} \wedge test(c_{\text{✘}}) = \text{✘}$ hold.
The goal is to find $c'_{\text{✘}} = ddmin(c_{\text{✘}})$ such that $c'_{\text{✘}} \subseteq c_{\text{✘}}$, $test(c'_{\text{✘}}) = \text{✘}$, and $c'_{\text{✘}}$ is 1-minimal.
The *minimizing Delta Debugging algorithm ddmin*(c) is

$$ddmin(c_{\text{✘}}) = ddmin_2(c_{\text{✘}}, 2) \quad \text{where}$$

$$ddmin_2(c'_{\text{✘}}, n) = \begin{cases} ddmin_2(\Delta_i, 2) & \text{if } \exists i \in \{1, \ldots, n\} \cdot test(\Delta_i) = \text{✘ ("reduce to subset")} \\ ddmin_2\big(\nabla_i, \max(n-1, 2)\big) & \text{else if } \exists i \in \{1, \ldots, n\} \cdot test(\nabla_i) = \text{✘ ("reduce to complement")} \\ ddmin_2\big(c'_{\text{✘}}, \min(|c'_{\text{✘}}|, 2n)\big) & \text{else if } n < |c'_{\text{✘}}| \text{ ("increase granularity")} \\ c'_{\text{✘}} & \text{otherwise ("done").} \end{cases}$$

where $\nabla_i = c'_{\text{✘}} - \Delta_i$, $c'_{\text{✘}} = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n$, all Δ_i are pairwise disjoint, and $\forall \Delta_i \cdot |\Delta_i| \approx |c'_{\text{✘}}|/n$ holds.
The recursion invariant (and thus precondition) for $ddmin_2$ is $test(c'_{\text{✘}}) = \text{✘} \wedge n \leq |c'_{\text{✘}}|$.

Figure 7.1: 1-Minimal Algorithm

This algorithm is quite intuitive. Let's break it down step by step.

The `ddmin()` function kicks off the process with a fixed granularity parameter of '2'. This '2' dictates the initial granularity, meaning the algorithm first divides the input into segments that are half the size of the original input.

The `ddmin2()` function's initial task is to split the input into delta and complement sets. After this, we iterate through all deltas and complements. We'll start by executing the delta set.

If a failure is detected while testing with a delta set, the function recursively calls itself with the current delta and a granularity of 2 (the same as the initial case).

If a failure is found during testing with the complement set, it recursively calls itself with the current complement set and `max(n - 1, 2)`. This means the granularity is slightly adjusted if n is greater than 2, but the minimum granularity remains 2, mirroring the first case.

Finally, if no error is found in either the delta or complement set, the algorithm attempts to increase the granularity to explore more test cases.

The algorithm stops if the function flow doesn't fall into any of these three conditions.

7.5 Implementation

We start implement `ddmin()` as a library first and test. Once we understood by implementing the algorithm with library first, and we will port it into the fuzzer.

```rust
// delta_debugging/src/lib.rs
```

```
pub type Data = Vec<u8>;

#[derive(PartialEq, Debug)]
pub enum TestResult {
    Pass,
    Fail,
}

pub type TestFn = Box<dyn Fn(&Data) -> TestResult>;

pub fn ddmin(data: &Data, test: TestFn) -> Data {
    do_ddmin(data, 2, test)
}

fn do_ddmin(data: &Data, n: usize, test: TestFn) -> Data {
    let (delta_set, complement_set) = split(data, n);
    for delta in &delta_set {
        if test(delta) == TestResult::Fail {
            if delta.len() == 1 {
                return delta.to_vec();
            }
            return do_ddmin(delta, 2, test);
        }
    }
    for complement in &complement_set {
        if test(complement) == TestResult::Fail {
            return do_ddmin(complement, max(n - 1, 2), test);
        }
    }
    if n < data.len() {
        return do_ddmin(data, min(data.len(), 2 * n), test);
    }
    data.to_vec()
}

/// Return delta and complement
pub fn split(data: &Data, n: usize) -> (Vec<Data>, Vec<Data>) {
    if n == 0 {
        return (Vec::new(), Vec::new());
    }
    let data_len = data.len();
```

```
    let exact_chunk_size = data_len / n;
    let remainder = data_len % n;

    let mut delta_boundaries = Vec::new();
    let mut cur_pos = 0;

    for i in 0..n {
        let mut chunk_size = exact_chunk_size;
        if i < remainder {
            chunk_size += 1;
        }
        if cur_pos + chunk_size > data_len {
            break;
        }
        delta_boundaries.push((cur_pos, cur_pos + chunk_size));
        cur_pos += chunk_size;
    }

    let delta_set: Vec<Data> = delta_boundaries
        .iter()
        .map(|&(start, end)| data[start..end].to_vec())
        .collect();

    let mut complement_set: Vec<Data> = Vec::new();
    for i in 0..delta_set.len() {
        let (start, end) = delta_boundaries[i];
        let mut complement_bytes_raw = Vec::new();
        if start > 0 {
            complement_bytes_raw.extend_from_slice(&data[0..start]);
        }
        if end < data_len {
            complement_bytes_raw.extend_from_slice(
              &data[end..data_len]);
        }
        complement_set.push(complement_bytes_raw);
    }
    let filtered_complements: Vec<Data> = complement_set
        .into_iter()
        .filter(|c| !delta_set.contains(c))
        .collect();
    (delta_set, filtered_complements)
```

```
84 }
```

The implementation closely mirrors the algorithm. The primary recursive call function is defined on line 16. Line 12 defines `ddmin2()`, which serves as the algorithm's entry function. It's worth noting that the `test` function acts as the real-world oracle, determining whether an input is failure-inducing. Therefore, we treat test as a first-class function that must be defined by the caller. The `split()` function returns both the delta and complement sets.

7.6 Testing

The following functions are for Delta Debugging (DD) testing, with multiple test cases. The first case uses the input demonstrated in the paper's working example.

Rust

```rust
// delta_debugging/tests/dd_test.rs

#[test]
fn test_ddmin() {
    let tcs = [
        (
            String::from("12345678"),
            String::from("178"),
            String::from("178"),
        ),
        (
            String::from("12345678"),
            String::from("1<78"),
            String::from("12345678"),
        ),
        (
            String::from("int a = 1; int b = 2; assert(a == b);"),
            String::from("assert(a == b);"),
            String::from("assert(a == b);"),
        ),
        (
            String::from("This is a test string with a problematic $
            character inside."),
            String::from("$"),
            String::from("$"),
        ),
        (
            String::from("function calculate_sum(a, b) { return a +
            b; print(\"calc\" }"),
            String::from("print(\"calc"),
            String::from("print(\"calc"),
```

```
        ),
        (
            String::from(
                "{'data': [1, 2, 3, 'value', {'key': 'nested'}, 'extra', {'bug': 'missing_brace']}",
            ),
            String::from("missing_brace'"),
            String::from("'missing_brace"),
        ),
    ];
    for tc in tcs {
        let (input, fail_inducing_input, expected) = (tc.0, tc.1, tc.2);
        test(input, fail_inducing_input, expected.into_bytes());
    }
}

fn test(input: String, fail_inducing_input: String, expected: Data) {
    let oracle = make_oracle(fail_inducing_input.into_bytes());
    let minimized = ddmin(&input.into_bytes(), oracle);
    println!(
        " {:?} == {:?}",
        byte_to_str(&minimized),
        byte_to_str(&expected)
    );
    assert_eq!(minimized, expected);
}

fn byte_to_str(data: &Data) -> String {
    String::from_utf8(data.to_vec()).unwrap()
}

fn make_oracle(fail_inducing_input: Data) -> Box<dyn Fn(&Data) -> TestResult> {
    Box::new(move |data: &Data| {
        let mut fail_inducing_cnt = HashMap::new();
        for v in &fail_inducing_input {
            *fail_inducing_cnt.entry(v).or_insert(0) += 1;
        }
        let mut input_cnt = HashMap::new();
        for v in data {
```

```
            *input_cnt.entry(v).or_insert(0) += 1;
        }
        for (key, &needed) in &fail_inducing_cnt {
            let available = input_cnt.get(key).copied().unwrap_or(0);
            if available < needed {
                println!("pass: {:?}", byte_to_str(data));
                return TestResult::Pass;
            }
        }
        println!("failured: {:?}", byte_to_str(data));
        TestResult::Fail
    })
}
```

The test input comprises: the original long failure-inducing input, its exact failure-inducing answer, and the expected 1-minimal output. We've defined an oracle that returns `TestResult::Pass` if the given input contains the failure-inducing elements, and `TestResult::Fail` otherwise.

If readers wish to observe the step-by-step progress, they can run the test with the command: `cargo test test_ddmin -- --exact --nocapture`.

7.7 Fuzzer Integration

Having studied and implemented the 1-minimal algorithm, it's time to integrate it into our fuzzer. Let's first outline the necessary steps:

- **Oracle Definition**: Our fuzzer's campaign already serves as the oracle. The oracle, in this context, is the fuzzing process itself, testing the input that's being minimized by the 1-minimal procedure.
- **Seed Minimization**: When a crash-inducing seed is found, we enter a minimization loop—the 1-minimal procedure. This pauses the generation of new seeds. Once the minimized seed is identified, we store it in the crash directory and then resume the original fuzzing process.

Rust

```rust
// fuzzer/src/fuzzer.rs

fn is_crash(&self, status: i32) -> bool {
    status != PROCESS_EXIT_NORMAL as i32 && status != SIGKILL
}

fn oracle(
    &mut self,
    child_stdin: &mut Option<ChildStdin>,
    timeout: Duration,
    seed: &Seed,
```

```
) -> Result<TestResult> {
    self.feed_seed(child_stdin, seed)?;
    self.wakeup_forkserver(HostSend::Wakeup(timeout.as_secs()));
    let status = self.wait_forkserver();
    self.clear_new_coverage();
    self.clear_visited_edges();
    if self.is_crash(status) {
        Ok(TestResult::Fail)
    } else {
        Ok(TestResult::Pass)
    }
}

fn ddmin(
    &mut self,
    child_stdin: &mut Option<ChildStdin>,
    timeout: Duration,
    seed: &Seed,
) -> Result<Seed> {
    self.do_ddmin(child_stdin, timeout, seed, 2)
}

fn do_ddmin(
    &mut self,
    child_stdin: &mut Option<ChildStdin>,
    timeout: Duration,
    seed: &Seed,
    n: usize,
) -> Result<Seed> {
    let (delta_set, complement_set) = split(&seed.get_input().to_vec(), n);
    for delta in &delta_set {
        let delta_seed = Seed::new(delta.to_vec(), 0);
        if let Ok(test_result) = self.oracle(child_stdin, timeout, &delta_seed) {
            if test_result == TestResult::Fail {
                if delta.len() == 1 {
                    return Ok(delta_seed);
                }
                return self.do_ddmin(child_stdin, timeout, &delta_seed, 2);
            }
```

```
            }
        }
        for complement in &complement_set {
            let complement_seed = Seed::new(complement.to_vec(), 0);
            if let Ok(test_result) = self.oracle(child_stdin, timeout, &complement_seed) {
                if test_result == TestResult::Fail {
                    return self.do_ddmin(child_stdin, timeout, &complement_seed, max(n - 1, 2));
                }
            }
        }
        let seed_len = seed.get_input().len();
        if n < seed_len {
            return self.do_ddmin(child_stdin, timeout, seed, min(seed_len, 2 * n));
        }
        Ok(seed.clone())
}
```

```rust
// fuzzer/src/fuzzer.rs

pub fn run(&mut self, program_file: &str) -> Result<FuzzResult> {
    ...
    ...
    // crash found
    if self.is_crash(status) {
        if let Ok(minimized) = self.ddmin(&mut child_stdin, timeout, &seed) {
            self.crashes.insert(minimized.clone());
            minimized.to_file(self.crashes.len());
            self.send(FuzzShot::Crash(CrashInfo::new(
                self.crashes.len(),
                seed,
                minimized,
            )));
        }
    }
    ...
}
```

The `oracle()` function utilizes the fuzzing procedure as an oracle. The oracle result is then passed to the 1-minimal procedure for further minimization.

7.7.1 Homework

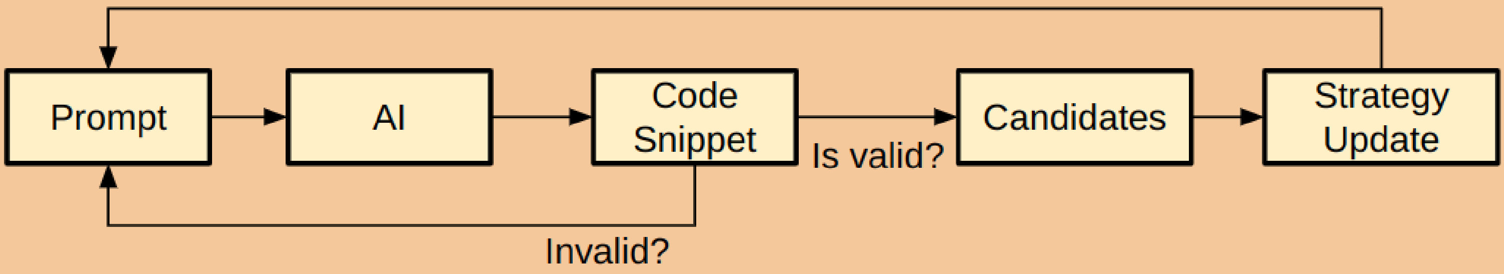
Exercise 7.1 We suggest the following topics for further study and exploration:

- **Optimization**: Readers can explore state-of-the-art papers for more optimized solutions to find minimized failure-inducing inputs.

Part VII

LLM-based Synthesis

“Leverage an LLM to build a program that generates programs”



8. LLM-based Synthesis

8.1 Hello, AI

The ubiquity of AI is no longer a major hurdle. Across almost every field, interactive Large Language Model (LLM [30]) are widely used and provide excellent assistance for daily tasks.

My first real experience with AI was in 2019. At that time in South Korea, there were very few AI lectures offered in university courses. Computer vision was the main topic for AI integration, and Convolutional Neural Network (CNN [31]) were a hot topic, with the MNIST [32] classification problem often serving as a starting point. Simultaneously, in the academic world, Natural Language Processing (NLP [33]) was a hot topic, with models like Long Short-Term Memory (LSTM [34]) being a primary example for tasks like language translation.

Back then, I didn’t expect AI to become so dominant, especially not how strongly it would become tied to our daily lives.

Nowadays, when a new keyword or concept comes to my attention, I prefer to ask an LLM rather than searching on Google. When I used to Google, I had to expend a lot of effort to filter out correct from incorrect information, identify the key points, understand the motivation, and grasp the output. Since I lacked the foundational knowledge (which was why I was searching in the first place), this process was very time-consuming. When I first started asking LLMs, I was surprised that the answers to my questions seemed correct and directly addressed the major concepts I was curious about. Moreover, the interactive nature of LLMs allows me to ask follow-up questions. It’s fascinating to reflect on how AI has integrated into daily life, much like the experience I’ve shared.

As a senior CS student in 2019, I didn’t anticipate this success. Today, I sometimes wonder what the next hot, winning, and game-changing technology will be. Personally, I believe the AI model itself is not the only important part. I now look at the entire AI ecosystem to see which parts can be improved, where I can contribute, which parts are connected to computer science (software), and what seems interesting and fun.

My current and graduate majors were not in AI. However, learning about other fields has led to new discoveries in my personal research, provided a new perspective, and reignited the joy of programming. So, why not take some time today to explore an AI-related product, technology, or trend? It can also be beneficial to research other fields, even if the keyword isn't "AI," and see a different area from your daily work. It may give you a perspective you couldn't have imagined before.

In this chapter, our goal is to implement an **LLM-based synthesis tool** to generate test code for our implementation.

8.2 Introduction to LLM-based Fuzzer

Academic research is actively exploring how to integrate LLMs. For example, a referenced paper on an LLM-based fuzzer, [35], successfully found new, previously undiscovered bugs. In my opinion, this type of fuzzer is particularly effective at finding bugs in new features. A prime use case would be testing new syntax or libraries of a programming language, new API usages, and so on. This fuzzer typically consists of two main components.

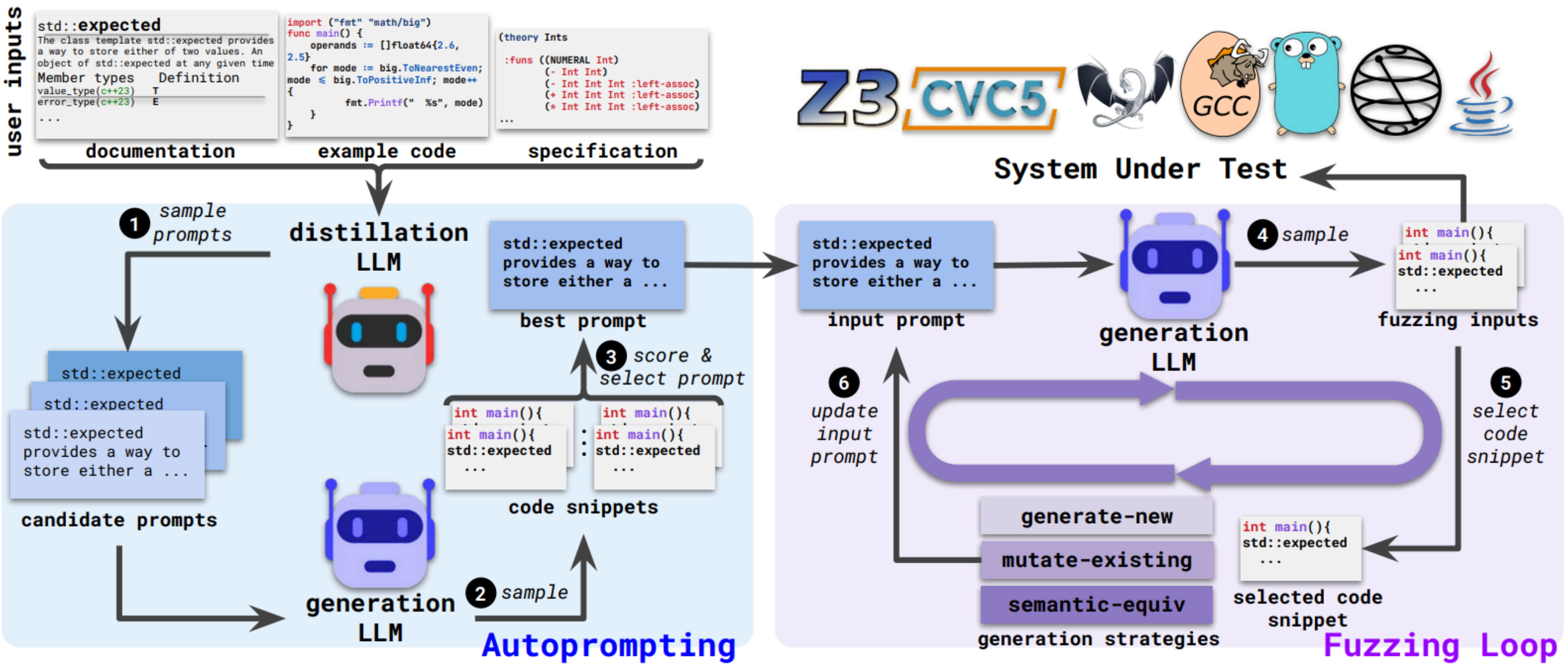


Figure 8.1: Overview of Fuzz4All

The architecture, which is based on a figure from the paper, has two major components: **Autoprompting** and the **Fuzzing Loop**.

8.2.1 Autoprompting

Autoprompting is a one-time initialization task that runs before the fuzzing loop begins. It takes documentation, example code, and specifications as input. These inputs are fed to an LLM, which is guided to generate a concise, abstracted set of statements. A scoring function then assigns scores to these statements based on various metrics, such as code coverage or bug-finding potential. The paper's approach specifically assigns a higher score if a generated code snippet is accepted and runnable by the target System Under Test (SUT). This process ultimately yields a set of high-quality candidate prompts to be used in the fuzzing loop.

Algorithm 1: Autoprompting for fuzzing

```
Function Autoprompting:
    Input  : userInput, numSamples
    Output : inputPrompt

    greedyPrompt ← M_D (userInput, APInstruction, temp=0)
    candidatePrompts ← [ greedyPrompt ]
    while |candidatePrompts| < numSamples do
        prompt ← M_D (userInput, APInstruction, temp=1)
        candidatePrompts ← candidatePrompts + [ prompt ]
    inputPrompt ← argmax_{p∈candidatePrompts} Scoring (M_G (p), SUT)
    return inputPrompt
```

Figure 8.2: AutoPrompting

The autoprompting algorithm, which is taken from the paper, is described above.

It begins by generating an initial “greedy” prompt from a distillation model with a temperature of zero. This setting makes the output highly conservative. The process continues until a sufficient number of candidate prompts have been collected.

To explore a wider range of possibilities, the algorithm then generates additional prompts using the same format but with the temperature set to one. This encourages more creative and diverse outputs. Finally, the best prompt is selected using the `argmax()` function, which finds the highest-scoring candidate based on the defined scoring function.

8.2.2 Fuzzing Loop

The prompt generated during the Autoprompting phase is fed into a generation LLM, which yields a code snippet based on the input. This generated code snippet is then tested against the SUT. Mutation is a key part of this process, and there are three mutation strategies that we will detail shortly. This entire loop is repeated until the given time budget is exhausted.

Algorithm 2: Fuzzing loop

```
Function FuzzingLoop:
    Input  :inputPrompt, timeBudget
    Output:bugs
    genStrats ← [ generate-new, mutate-existing,
      semantic-equiv ]
    fuzzingInputs ← M_G (inputPrompt + generate-new)
    bugs ← Oracle (fuzzingInputs, SUT)
    while timeElapsed < timeBudget do
        example ← sample (fuzzingInputs, SUT)
        instruction ← sample (genStrats)
        fuzzingInputs ← M_G (inputPrompt + example +
          instruction)
        bugs ← bugs + Oracle (fuzzingInputs, SUT)
    return bugs
```

Figure 8.3: Fuzzing Loop

As described in the algorithm, there are three mutation strategies: new-generation, mutate-existing, and semantic-equivalent.

The new-generation strategy involves a prompt that creates a program from scratch. This is the strategy used to start the initial code snippet generation (Also, being used in the loop continuously). During the fuzzing loop, a strategy is selected randomly.

The mutate-existing strategy prompts the LLM to generate a mutated code snippet based on a previous generation. The semantic-equivalent strategy, on the other hand, guides the LLM to create a semantically equivalent code snippet. The generated code snippets from these strategies are fed to the SUT, and the process is repeated.

This scheme is straightforward to understand, yet its effectiveness is remarkable. It has successfully found bugs in compilers like GCC and Golang, as detailed in [36]. Readers interested in the paper's experiments are encouraged to read the full document.

In summary, we have studied the power and motivation of LLMs, how they are integrated into fuzzing, and how they are used. Inspired by this, we will implement an LLM-based synthesis tool to generate test code snippets for our OOB detector (from Chapter 2, ASan).

8.3 Introduction to Program Synthesis

Program synthesis [37] is the task of automatically generating a program from a high-level specification. The synthesis tool takes a requirement and attempts to generate code that satisfies it. For example, to create a C function that adds two integers, our specification would be the function signature: int add(int, int);.

The main challenge is implementing the function body. To do this, the synthesizer needs a formal grammar or a set of rules it can use to construct and combine code elements. For instance:

Spec

```
a
b
```

```
+
-
return
;
```

Based on this grammar, the synthesizer will try to generate code snippets such as:

```
a b +
a b -
a return b
...
...
```

To minimize the generation of invalid syntax, we can guide the synthesizer with defined constraints. For example: The variables `a` and `b` must be located between an operator, such as `+` or `-`. The last statement should start with `return`.

Finally, we should provide input-output examples such as:

Input-output

```
examples
1 2 3
2 3 5
...
```

With these more specific guides, the synthesizer will eventually generate the intended function, `int add(int a, int b) { return a+b; }`. However, it might also generate semantically equivalent but more complex code, such as `int add(int a, int b) { return a+b-a-b+a+b; }`. To avoid generating unnecessary code and save time, branch pruning is needed. If you are interested in traditional program synthesis, I recommend exploring university courses.

8.4 LLM-based Synthesis

This chapter's main topic is the implementation of a synthesizer that generates test programs for our OOB detector. This tool can be extended to generate other programs that target specific types of bugs.

As we saw, traditional program synthesis finds it difficult to generate even a simple function like "add." In contrast, our tool will generate more complex programs that contain two kinds of bugs: "buffer overflow" and "use-after-free".

The synthesizer is implemented in just 114 lines of code. For this model, we've removed the Autoprompting phase and replaced it with a hand-written initial prompt, which serves similarly as an output of Autoprompting. Our initial prompt is as follows:

```
Requirements:
- Do not use any file-related functionality.
- Only use standard library functions: malloc, free, and strcpy.
- Use the function signature: void main().
- Add a brief comment next to the buggy line indicating the type of bug.
```

```
- Provide only one function, with no additional explanation or
output.
- **Include only one bug type per generated code.**
```

Here are the constraints we used to guide the LLM:

- Security: For security reasons, we instructed the LLM not to use any file-related functionality or system calls.
- Standard Library: We guided the LLM to use standard library functions such as `malloc`, `free`, and `strcpy`.
- Main Function: The main function's signature should be `void main()`.
- Bug Indicators: The LLM is instructed to add a brief comment indicating the intended bug type on the line where the buggy line is instrumented.
- Conciseness: The generated code should be concise, containing only a single function with no detailed comments per line.
- Single Bug Type: Each generated code snippet must contain only one bug type and should not mix two kinds of bugs simultaneously.

Here are the mutation strategy prompts (we use the same three rules, but different prompts):

- `new-generation-bof`: "Please create a program that triggers buffer overflow bug."
- `new-generation-uaf`: "Please create a program that triggers use-after-free bug."
- `mutate-existing`: "Please create a mutated program that modifies the previous generation."
- `sementic-equiv`: "Please create a semantically equivalent program to the previous generation."

Here is the complete synthesizer code:

```python
//synthesis/asan_model.py

from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
    BitsAndBytesConfig,
    StoppingCriteriaList,
    StoppingCriteria,
)
import torch
import random
import re
import os
import subprocess

def extract_first_code_block(text):
    pattern = r'\`\`\`(?:\w+)?\n?(.*?)\`\`\`'
    match = re.search(pattern, text, re.DOTALL)
```

```python
    if match:
        return match.group(1).strip()
    return ""

def gen_model():
    model_name = "Qwen/Qwen2.5-Coder-7B-Instruct"
    quantization_config = BitsAndBytesConfig(
        load_in_4bit=True,
        bnb_4bit_compute_dtype=torch.float16,
        bnb_4bit_quant_type="nf4",
        bnb_4bit_use_double_quant=True,
    )
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    model = AutoModelForCausalLM.from_pretrained(
        model_name,
        quantization_config=quantization_config,
        device_map="auto",
        trust_remote_code=True,
    )
    print(f"Model loaded: {torch.cuda.memory_allocated()/1024**3:.2f}GB")
    return model, tokenizer

def gen_prompt(tokenizer, model_device, user_prompt):
    messages = [
        {"role": "system", "content": "You are a helpful coding assistant"},
        {"role": "user", "content": user_prompt}
    ]
    text = tokenizer.apply_chat_template(messages, tokenize=False, add_generation_prompt=True)
    return tokenizer([text], return_tensors="pt").to(model_device)

def execute(model, tokenizer, input_prompt):
    with torch.no_grad():
        generated_ids = model.generate(
            **input_prompt,
            max_new_tokens=256,
            do_sample=True,
            repetition_penalty=1.0,
            temperature=1.0,
            top_p=0.7,
```

```
            pad_token_id=tokenizer.eos_token_id,
        )
        generated_ids = [output_ids[len(input_ids):] for input_ids, output_ids in zip(input_prompt.input_ids, generated_ids)]
        response = tokenizer.batch_decode(generated_ids, skip_special_tokens=True)[0]
        return response

def is_valid_code(c_code):
    result = subprocess.run(
        ['clang', '-x', 'c', '-'],
        input=c_code,
        text=True,
        capture_output=True
    )
    if result.returncode == 0:
        os.remove("./a.out")
    return result.returncode == 0

def gen_example(n_samples=5):
    model, tokenizer = gen_model()
    guide_prompt = '''
        Requirements:
        - Do not use any file-related functionality.
        - Only use standard library functions: malloc, free, and strcpy.
        - Use the function signature: void main().
        - Add a brief comment next to the buggy line indicating the type of bug.
        - Provide only one function, with no additional explanation or output.
        - **Include only one bug type per generated code.**
        '''
    newgen_bof_prompt = "Please create a program that triggers buffer overflow bug."
    newgen_uaf_prompt = "Please create a program that triggers use-after-free bug."
    se_prompt = "Please create a semantically equivalent program to the previous generation"
    mutate_prompt = "Please create a mutated program that modifies the previous generation"
    gen_stats = [newgen_bof_prompt, newgen_uaf_prompt, mutate_prompt, se_prompt]

```

```
92      # generated mutated fuzzing inputs
93      fuzzing_inputs = []
94      while len(fuzzing_inputs) < n_samples:
95          instruction = random.choice(gen_stats)
96          print(f"generating ...{len(fuzzing_inputs)}")
97          print(f"selected instruction: {instruction}")
98          prompt_id = gen_prompt(tokenizer, model.device, guide_prompt + "\n" + instruction)
99          fuzzing_input = execute(model, tokenizer, prompt_id)
100         fuzzing_input = extract_first_code_block(fuzzing_input)
101         if is_valid_code(fuzzing_input):
102             fuzzing_inputs.append(fuzzing_input)
103         else:
104             print(f"compile failed: \n {fuzzing_input}")
105     return fuzzing_inputs
106
107 def write_c_files(fuzzing_inputs):
108     os.makedirs("generated", exist_ok=True)
109     for i, code in enumerate(fuzzing_inputs):
110         filename = f"generated/{i + 1}.c"
111         with open(filename, 'w') as f:
112             f.write(code)
113
114 n_samples = 5
115 examples = gen_example(n_samples)
116 write_c_files(examples)
```

Note on line 17: We added an escape character '\' in this code block because the character overlaps with the book's code annotation. This character is not present in the original code.

We use the 'Qwen/Qwen2.5-Coder-7B-Instruct' model [38]. Its 7B parameters are a reasonable scale to be executed on a home laptop with quantization enabled.

The `gen_model()` function initializes the model and tokenizer and sets up quantization. Model quantization converts the type of the model's weights to a lower bit, which makes it faster and more lightweight at the trade-off of a slight degradation in accuracy.

The `execute()` function invokes inference with a temperature of 1.0, limits the token size to 256, and returns the generated code snippet.

The `gen_example()` function is the main loop. It takes a number of samples, which specifies how many code snippets to generate. First, it defines the prompts for guidance and mutation. In the for loop, it randomly selects one of the mutation strategies, executes the prompt, and checks if the generated code is a valid program through the `is_valid_code()` function. Finally, the generated code is stored to disk via the `write_c_files()` function.

Let's run the synthesizer (Author's GPU spec is NVIDIA GeForce RTX 4060 Laptop GPU)

```shell
> python3 asan_model.py
Loading checkpoint shards: 100%|██████████████████████████████████████████████████
4/4 [00:16<00:00,  4.05s/it]
Model loaded: 5.19GB
generating ...0
selected instruction: Please create a program that triggers use-
after-free bug.
generating ...1
selected instruction: Please create a semantically equivalent
program to the previous generation
generating ...2
selected instruction: Please create a semantically equivalent
program to the previous generation
compile failed:
 #include <stdlib.h>
#include <string.h>

void main() {
    char *str1 = (char *)malloc(20 * sizeof(char)); // Allocate
    memory for string
    char *str2 = (char *)malloc(20 * sizeof(char)); // Allocate
    memory for string

    if (str1 == NULL || str2 == NULL) {
        // Handle memory allocation failure
        return;
    }

    strcpy(str1, "Hello, World!"); // Copy string to str1
    strcpy(str2, str1); // Copy str1 to str2

    // Buggy line: Memory leak
    free(str1); // Free str1
    // str2 is not freed, causing a memory leak

    // Buggy line: Use of uninitialized pointer
    printf("%s\n", str2); // Use of str2, which is not freed

    free(str2); // Free str2
}
generating ...2
```

```
selected instruction: Please create a program that triggers buffer overflow bug.
generating ...3
selected instruction: Please create a program that triggers buffer overflow bug.
generating ...4
selected instruction: Please create a program that triggers use-after-free bug.
```

If a generated code snippet is not compilable, the output will show the code like the one above. The reason for the failure in this case is the missing declaration of #include <stdio.h>. The full error message is shown below:

failure.c:21:5: note: include the header <stdio.h> or explicitly provide a declaration for 'printf'.

A directory named 'generated' will be created, containing five C files: 1.c, 2.c, 3.c, 4.c, and 5.c. Let's examine the contents of 1.c.

```c
1  #include <stdlib.h>
2  #include <string.h>
3
4  void main() {
5      char *str = (char *)malloc(20 * sizeof(char)); // Allocate memory
6      if (str == NULL) {
7          exit(1);
8      }
9      strcpy(str, "Hello, World!"); // Copy string to allocated memory
10
11     free(str); // Free the allocated memory
12
13     // Use-after-free bug: str is used after being freed
14     strcpy(str, "This will cause a use-after-free error"); // Buggy line
15 }
```

The generated content may differ with each run. This file, for example, contains 15 lines and UAF bug on line 14.

This demonstrates that we can effectively utilize the LLM-based synthesizer for testing our OOB detector. Rather than writing test cases by hand, we can simply let the LLM generate them for us.

In our daily lives, we regularly use LLM services such as GPT and Gemini. The number of parameters for GPT-3.5 is known to be 175B, while our selected Qwen model has 7B, a difference of approximately 25x. The scale of a 175B model makes it difficult to host on a home laptop due to GPU memory capacity.

Another significant difference is inference speed. You may have noticed that the synthesizer we built takes some time to generate tokens, even with the 256-token length constraint. In contrast, GPT and Gemini respond almost instantly. This suggests that their inference servers are equipped with extremely powerful and expensive hardware.

8.4.1 Homework

Exercise 8.1 We suggest the following topics for further study and exploration:

- **Topic #1: Generating Other Types of Bugs with the LLM**
  - Readers can modify the prompt to generate other types of bugs they are interested in.
- **Topic #2: Exploring the LLM-based Synthesizer**
  - The selected LLM is a specialized model for coding tasks. You can extend its use to synthesize code for other programming challenges, such as those in software engineering. Consider what tasks in your daily work could be synthesized and automated. For further ideas, you can also refer to state-of-the-art papers.

8.5 Retrospect

When I study program synthesis (Programming By Example; PBE), the area was really interested and it was fun. PBE takes pairs of input and output and grammar rules and it generates the program, especially source code. Thus, program makes program. The most popular use case of PBE is Flash Fill, which recorgnize the pattern of excel sheet and fills empty cells where the Flash Fill [39] inserts.

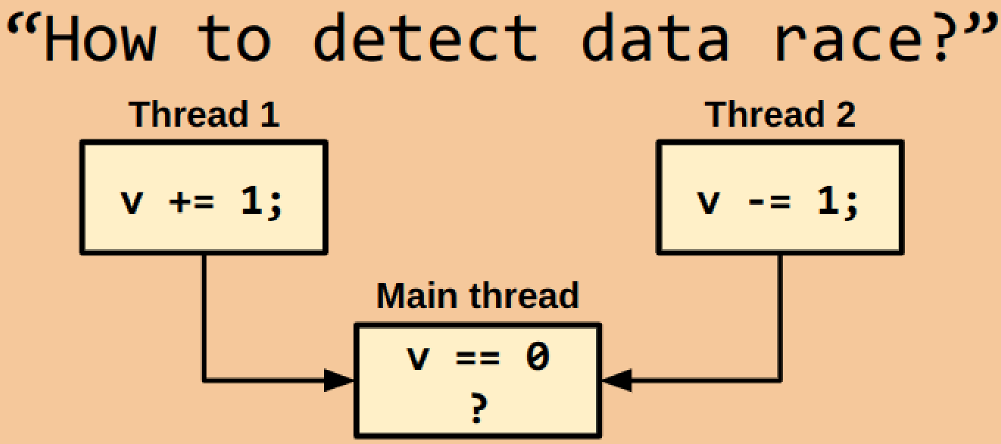
After the emergence of generative LLMs, coding tasks have become one of their major strengths. I initially thought that this would make traditional program synthesis obsolete, but I have completely reversed that opinion. The significant difference between the two is that a traditional program synthesizer yields a deterministic output for a given input (or a semantically equivalent one), whereas an LLM does not. If you ask an LLM the same prompt twice, the answer will be different, and sometimes the semantic meaning will be entirely different as well.

In this sense, I believe that guaranteeing a deterministic output is still a requirement in certain environments, such as mission-critical applications. Due to the unexplainability of an LLM's inference behavior, a traditional algorithmic approach remains attractive. I personally have concerns that the unpredictable and uncontrollable nature of LLM responses could present significant hurdles in some areas.

Therefore, it is crucial to first inspect an application's characteristics to determine if it is suitable for leveraging the power of LLMs, or if a more conservative, deterministic approach is required.

Part VIII

Data Race Detector



9. Introduction to Data Race

9.1 Introduction

Over time, hardware processor performance has steadily improved. One of the major factors driving this is the increase in the number of CPU cores. With two cores, a processor can execute two tasks simultaneously; with four cores, it can handle four tasks at once. If a processor is limited to a single core, no parallelism is possible, and all tasks must be executed sequentially.

Today, nearly all commercial products are equipped with multi-core CPUs. For example, my laptop has 32 cores. While performance is significantly enhanced by having many cores, their effective utilization presents another problem. A key challenge is a data race. **A data race is a bug where multiple threads access a shared memory location concurrently, with at least one access being a write, leading to unpredictable results.** Let’s consider an example below.

```c
// example-code/race/race-example.c

#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

volatile int counter = 0;
int n = 20000;

void* increment_counter(void* arg) {
    for (int i = 0; i < n; i++) {
        counter++;
    }
    return NULL;
}

```

```
int main() {
    pthread_t thread1, thread2;

    if (pthread_create(&thread1, NULL, increment_counter, NULL) != 0) {
        perror("Error creating thread 1");
        return 1;
    }
    if (pthread_create(&thread2, NULL, increment_counter, NULL) != 0) {
        perror("Error creating thread 2");
        return 1;
    }
    if (pthread_join(thread1, NULL) != 0) {
        perror("Error joining thread 1");
        return 1;
    }
    if (pthread_join(thread2, NULL) != 0) {
        perror("Error joining thread 2");
        return 1;
    }

    printf("Expected counter value: %d\n", 2 * n);
    printf("Actual counter value: %d\n", counter);

    return 0;
}
```

The expected output of this example is 40000, but a data race means the program is not guaranteed to yield this result every time it runs. If you don't immediately see a different value, try running the program multiple times; you should quickly observe an incorrect output.

Note 9.1 **Volatile keyword**

Note the use of the `volatile` [40] keyword. We added it to this example to ensure the data race is consistently observable. Without `volatile`, a compiler might make optimizations—such as assuming that the value of counter won't change unexpectedly—and effectively remove the loop, thus hiding the data race. The `volatile` keyword prevents the compiler from making such assumptions, forcing it to read the variable from memory on every access. This keyword is typically used for variables that can be modified by factors external to the current thread, such as hardware registers or other threads.

As we have seen, data races are a common and notorious bug in multi-threaded programming. A data race often does not cause an explicit crash; instead, it corrupts the value of a shared variable. This corruption can then propagate through the program, leading to unpredictable behaviors that are notoriously difficult to debug.

Let's use a Rust example.

```rust
// example-code/race/example-race.rs

use std::thread;

static mut COUNTER: i32 = 0;

fn main() {
    let mut handles = vec![];

    for _ in 0..2 {
        let handle = thread::spawn(|| {
            for _ in 0..20000 {
                COUNTER += 1;
            }
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Final counter value: {}", COUNTER);
}
```

The semantic meaning is the same as the C example, but this program is not compilable. The following error will be produced:

```shell
error[E0133]: use of mutable static is unsafe and requires unsafe function or block
  --> race-example.rs:11:17
   |
11 |                 COUNTER += 1;
   |                 ^^^^^^^ use of mutable static
   |
   = note: mutable statics can be mutated by multiple threads: aliasing violations or data races will cause undefined behavior

```

```
error[E0133]: use of mutable static is unsafe and requires unsafe function or block
  --> race-example.rs:21:41
   |
21 |     println!("Final counter value: {}", COUNTER);
   |                                         ^^^^^^^ use of mutable static
   |
   = note: mutable statics can be mutated by multiple threads: aliasing violations or data races will cause undefined behavior

error: aborting due to 2 previous errors

For more information about this error, try `rustc --explain E0133`.
```

A simple global variable is considered unsafe in Rust and will not compile. Similarly, some modern compilers impose constraints on language expressions to produce a safe binary at compile time. To make this code compilable, you'll need to wrap it in an `unsafe` block.

```rust
// example-code/race/example-race-unsafe.rs

use std::thread;

static mut COUNTER: i32 = 0;

fn main() {
    let mut handles = vec![];

    for _ in 0..2 {
        let handle = thread::spawn(|| unsafe {
            for _ in 0..20000 {
                COUNTER += 1;
            }
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    unsafe {
```

```
        println!("Final counter value: {}", COUNTER);
    }
}
```

Adding the `unsafe` keyword allows Rust to compile the program, but it shifts the responsibility for potential issues to the developer. When run, the program produces an unexpected result, not 40000.

In the previous section, we were introduced to the data race problem and saw how two compilers—C and Rust—handle it. C ignores data races at compile time, whereas Rust's ownership system is designed to prevent them.

The goal of this chapter is to develop a data race detector based on the seminal paper by [41].

9.2 Theory

Several seminal papers address data race detection. We will introduce two of them, and our implementation will be based on the latter paper.

9.2.1 Happens-Before

Nodes in a distributed system communicate by sending and receiving messages. We will demonstrate an example of a defect that can arise in the context of event ordering during this communication.

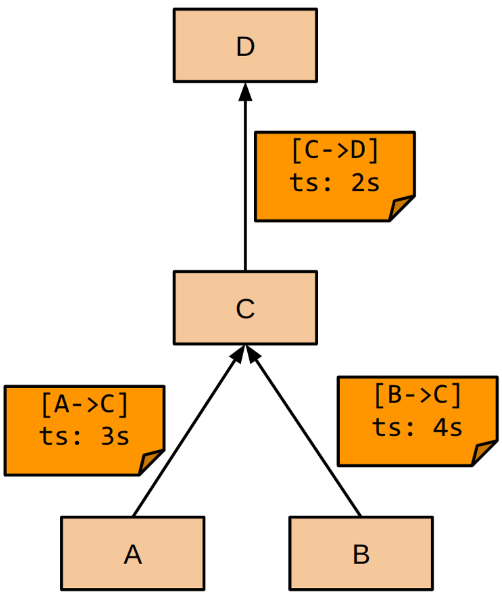


Figure 9.1: A and B send message to C, C sends to D. (ts = timestamp)

We begin with the fundamental principle that the local clocks of computers in a distributed system are never perfectly synchronized.

Consider a scenario where Node A and Node B send messages to Node C. After receiving both, Node C sends a message to Node D.

1. Node A sends its message at 3s.

2. Node B sends its message at 4s.

Let's assume Node C's local clock runs faster than Node A's and Node B's. When C receives the messages, its clock might read a time that precedes the timestamps from A and B. This mismatch reveals a critical problem: physical timestamps do not always align with the logical event order.

The logical order of events is what truly matters for causality. In our example, the sequence is as follows:

1. Node A sends a message to C, or Node B sends a message to C. The order of these two specific events doesn't matter.
2. Node C sends a message to D.

This establishes a causality between the events. Event 1 "happened-before" Event 2. This concept, known as the happened-before relation [42], considers the logical flow of events, completely ignoring the physical clock timestamps.

This is the key to synchronizing event ordering in a distributed system. Regardless of clock mismatches, each node can determine the causal relationship between events. These event relations can be chained and ordered. In our example, there are two possible causal chains:

- (1) A sends to C -> (2) B sends to C -> (3) C sends to D
- (1) B sends to C -> (2) A sends to C -> (3) C sends to D

The 'happened-before' relation can be used to identify data races. Let's look at an example below.

```c
volatile int counter = 0;
int n = 20000;

void* increment_counter(void* arg) {
    for (int i = 0; i < n; i++) {
        counter++;
    }
    return NULL;
}

... thread 1 and 2 execute `increment_counter()`
```

The event ordering of the increment_counter function without a lock is as follows:

1. The local variable `i` is initialized to zero.
2. The global variable `counter` is incremented.

If we assume only a single thread exists, the `counter` will reach the expected value because all events are strictly ordered with no overlaps. However, in a multi-threaded environment, Event 2 can occur simultaneously across multiple threads, meaning the events are not logically ordered. This is a classic example of a data race.

Let's introduce a critical section using a mutex:

```c
pthread_mutex_t mutex;
void* increment_counter(void* arg) {
    pthread_mutex_lock(&mutex);
    for (int i = 0; i < n; i++) {
```

```
            counter++;
        }
        pthread_mutex_unlock(&mutex);
        return NULL;
}
```

In a single-threaded environment, the event order is defined as:

1. The global mutex acquires a lock.
2. The local variable i is initialized to zero.
3. The global counter is incremented.
4. The global mutex releases the lock.

Just like the previous example, all events are ordered in a single-threaded context. What happens with two threads?

1. Thread 1 or Thread 2 acquires the lock on the mutex. We'll call this the "active thread." The other thread becomes the "standby thread."
2. The active thread initializes the local variable i to zero.
3. The active thread updates the global counter.
4. The active thread releases the lock.
5. The standby thread acquires the lock.
6. The standby thread initializes the local variable i to zero.
7. The standby thread updates the global counter.
8. The standby thread releases the lock.

With the critical section, all events are ordered, and no data race can occur. This demonstrates that the happened-before relation is a powerful tool for reasoning about and detecting data races.

9.2.2 LockSet

The paper by [41] introduces the LockSet algorithm, which we will implement. This intuitive algorithm is straightforward to understand. We've included the algorithm and a corresponding example from the paper below.

Let $locks\_held(t)$ be the set of locks held by thread t.
For each v, initialize $C(v)$ to the set of all locks.
On each access to v by thread t,
  set $C(v) := C(v) \cap locks\_held(t)$;
  if $C(v) = \{\ \}$, then issue a warning.

Figure 9.2: Lockset Algorithm

The LockSet algorithm operates as described below. Let v be a globally shared variable, and let locks_held(t) be the set of locks currently held by a thread t.

- **Initialization**: For each globally shared variable `v`, a set `C(v)` is initialized to contain all possible locks.
- **On Access**: Each time the variable v is accessed, the set `C(v)` is updated through a set intersection operation with `locks_held(t)`.
- **Detection**: If the set `C(v)` becomes empty, a data race is detected and reported.

The following example demonstrates how the algorithm identifies a data race.

Program	*locks_held*	*C(v)*
	`{}`	`{mu1,mu2}`
`lock(mu1);`		
	`{mu1}`	
`v := v+1;`		
		`{mu1}`
`unlock(mu1);`		
	`{}`	
`lock(mu2);`		
	`{mu2}`	
`v := v+1;`		
		`{}`
`unlock(mu2);`		
	`{}`	

Figure 9.3: Example

The variable `v` is accessed by each thread (first three statements by thread1 and the other three statements are by thread2). The `locks_held` set is empty for the first time for each threadd and `C(v)` is initilized with two locks (`mu1` and `mu2`). In the first thread, after `lock(mu1)` statement, `locks_held` is updated to `{mu1}`. When `v := v+1` is executed, the `C(v)` is refined to `{mu1}` by intersection operation as described in the algorithm. After `unlock(mu1)`, the `locks_held` becomes empty and nothing changed against `C(v)`.

when `lock(mu2)` is executed, `locks_held` for thread2 is updated to `{mu2}`. After `v := v+1` is executed, the `C(v)` is refined to empty set because the `C(v)` was `{mu1}`, so the set intersectinon yield empty set. At this point, a report is created.

Without following the algorithm step by step, we can figure out that the example contains real data race problem. The shared variable `v` is not protected with single mutex. Two differmnt mutex does not make critical section for shared variable `v` and results in concurrent read and write can be perfomed against `v`.

If we change two statements `lock(mu2)` and `unlock(mu2)` to `lock(mu1)` and `unlock(mu1)` in the example, then no data race is reported because the intersection would yield `{mu1}` and no empty set.

Let's trace the LockSet algorithm for a shared variable `v`. The example code shows `v` being accessed by two threads.

Initially, `locks_held` is empty for both threads, and `C(v)` is initialized with two locks: `{mu1, mu2}`.

Thread 1

- `lock(mu1)`: The `locks_held` set for Thread 1 becomes `{mu1}`.
- `v := v+1`: The algorithm updates `C(v)` by performing an intersection: `C(v) = C(v) ∩ locks_held(t1)`. This refines `C(v)` from `{mu1, mu2}` to `{mu1}`.
- `unlock(mu1)`: The `locks_held(t1)` set becomes empty, and `C(v)` remains `{mu1}`.

Thread 2

- `lock(mu2)`: The `locks_held(t2)` set for Thread 2 becomes `{mu2}`.
- `v := v+1`: The algorithm performs another intersection: `C(v) = C(v) ∩ locks_held(t2)`. The current `C(v)` is `{mu1}`, and `locks_held(t2)` is `{mu2}`. The intersection of these two sets is empty. `C(v)` becomes `{}`.
- **Result**: At this point, the algorithm reports a data race because `C(v)` is now empty.

Even without a step-by-step trace of the algorithm, we can see that this example contains a data race. The shared variable `v` is not protected by a single, consistent mutex. Using two different mutexes (`mu1` and `mu2`) does not create a proper critical section for `v`, allowing concurrent reads and writes to occur.

If we were to change the `lock(mu2)` and `unlock(mu2)` statements to `lock(mu1)` and `unlock(mu1)`, no data race would be reported. In that case, the intersection operation would yield `{mu1}`, preventing `C(v)` from becoming empty.

To precisely detect a data race, the paper's authors introduced a more advanced algorithm: a state-transition model. The model consists of four states, as shown in the state-transition diagram:

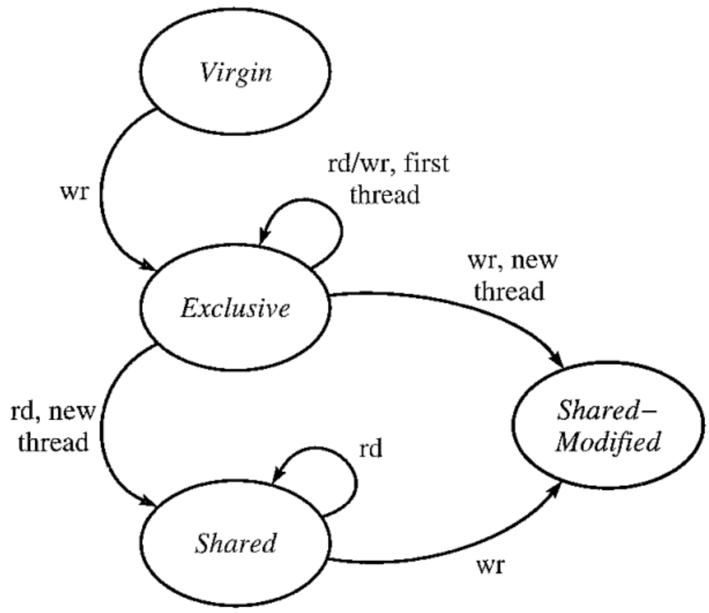


Figure 9.4: Example

- **Virgin**: This is the initial state, indicating that no operations have occurred yet.
- **Exclusive**: A state transitions to `Exclusive` from `Virgin` upon the first write operation. Subsequent read or write operations by the same thread do not change the state.
- **Shared**: A state transitions to `Shared` from `Exclusive` when another thread performs a read operation. A write operation by a different thread in this state will cause a transition to `Shared-Modified`.
- **Shared-Modified**: A data race is reported only in this state if the `C(v)` becomes empty.

Now it's time to implement this algorithm using an instrumentation module and a runtime.

9.2.3 Implementation

There are four tasks in the entire implementation roadmap. In our implementation, we target only static global variables. For dynamic data (heap), we will note it as homework at the end of this chapter.

- Initialization of `C(v)`: Implemented by static analysis
- Refinement of `C(v)`: Instrumented by module and runtime updates
- Update `lock_held(t)`: Instrumented by module and runtime updates
- State-transaction model: Implemented by runtime

9.2.3.1 Instrumentation

We introduce the entire instrumentation and static analysis loop first and explain step by step.

```rust
// instrument/src/race.rs

#[derive(Debug, PartialEq, Eq, Hash, Clone)]
pub enum LockTyp {
    Lock,
    UnLock,
}
impl LockTyp {
    pub fn i8(&self) -> i8 {
        match self {
            Self::Lock => 1,
            Self::UnLock => 0,
        }
    }

    pub fn from_i8(t: i8) -> Option<Self> {
        match t {
            1 => Some(Self::Lock),
            0 => Some(Self::UnLock),
            _ => None,
        }
    }
}

#[derive(Debug)]
pub enum AccessOperation {
    Write,
    Read,
}
impl AccessOperation {
```

```
    pub fn i8(&self) -> i8 {
        match self {
            Self::Write => 1,
            Self::Read => 0,
        }
    }

    pub fn from_i8(t: i8) -> Option<Self> {
        match t {
            1 => Some(Self::Write),
            0 => Some(Self::Read),
            _ => None,
        }
    }
}

#[derive(Debug, Clone)]
pub struct Lock<'ctx> {
    pub typ: LockTyp,
    pub lock: PointerValue<'ctx>,
}

impl<'ctx> Lock<'ctx> {
    fn new(typ: LockTyp, lock: PointerValue<'ctx>) -> Self {
        Self { typ, lock }
    }
}

impl<'ctx> PartialEq for Lock<'ctx> {
    fn eq(&self, other: &Self) -> bool {
        self.typ == other.typ && self.lock.as_value_ref() ==
        other.lock.as_value_ref()
    }
}

impl<'ctx> Eq for Lock<'ctx> {}

impl<'ctx> Hash for Lock<'ctx> {
    fn hash<H: Hasher>(&self, state: &mut H) {
        self.typ.hash(state);
        self.lock.as_value_ref().hash(state);
```

```
    }
}

#[derive(Default)]
pub struct RaceModule {}

impl InstrumentModule for RaceModule {
    fn instrument<'ctx>(
        &self,
        context: &'ctx Context,
        module: &Module<'ctx>,
        builder: &Builder<'ctx>,
    ) -> Result<()> {
        let global_int_vals = get_global_int_vals(module);

        let mut lock_candidate_set = HashMap::new();
        let mut live_locks = HashSet::new();
        let mut pthread_self_callsites = HashMap::new();
        let funcs: Vec<_> = module.get_functions().collect();
        for func in &funcs {
            // Skip funcs without bodies or those we've added
            if can_skip_instrument(&func) {
                continue;
            }
            for basic_blk in func.get_basic_blocks() {
                for instr in basic_blk.get_instructions() {
                    if let Some(lock) = get_lock(&instr) {
                        match lock.typ {
                            LockTyp::Lock => {
                                live_locks.insert(lock.clone());
                            }
                            LockTyp::UnLock => {
                                live_locks.remove(&lock);
                            }
                        }
                        // instrument for `lock_held(t)`
                        if let Some(next_instr) = instr.get_next_instruction() {
                            builder.position_before(&next_instr);
                        }
                        let thread_id = get_thread_id(
```

```
                        context,
                        module,
                        builder,
                        func,
                        &instr,
                        &mut pthread_self_callsites,
                    )?;
                    build_update_lock_held(context, module, builder, thread_id, lock)?;
                    continue;
                }

                if let InstructionOpcode::Load | InstructionOpcode::Store = instr.get_opcode() {
                    let (operand, access_op) = match instr.get_opcode() {
                        InstructionOpcode::Load => {
                            let operand = instr.get_operand(0)
                            .unwrap().left().unwrap();
                            (operand, AccessOperation::Read)
                        }
                        InstructionOpcode::Store => {
                            let operand = instr.get_operand(1)
                            .unwrap().left().unwrap();
                            (operand, AccessOperation::Write)
                        }
                        _ => unreachable!(),
                    };
                    let thread_id = get_thread_id(
                        context,
                        module,
                        builder,
                        func,
                        &instr,
                        &mut pthread_self_callsites,
                    )?;
                    handle_mem_access(
                        &instr,
                        &operand,
                        access_op,
                        &global_int_vals,
                        &mut lock_candidate_set,
```

```rust
                            &live_locks,
                            thread_id,
                            context,
                            module,
                            builder,
                        )?;
                    }
                }
            }
        }

        let constructor = build_race_init(context, module, builder, &lock_candidate_set)?;
        build_ctros(context, module, constructor)?;

        // Verify instrumented IRs
        module_verify(module)
    }
}
```

The first task is to collect all global integer variables. (This can be extended to other types later in the homework.)

```rust
// instrument/src/race.rs

fn get_global_int_vals<'ctx>(module: &Module<'ctx>) -> HashSet<PointerValue<'ctx>> {
    let globals: HashSet<PointerValue<'ctx>> = module
        .get_globals()
        .filter(|global| global.get_value_type().is_int_type())
        .map(|global| global.as_pointer_value())
        .collect();
    globals
}
```

In the main loop, we collect all global variables, initialize some data structures, and iterate through the functions.

Our first task is to identify whether a given instruction is a lock or unlock operation. The `get_lock()` function checks if the instruction is a call with two operands and if the function name is either `pthread_mutex_lock` or `pthread_mutex_unlock`. The mutex variable returned from this check is then inserted into `live_locks`, which helps initialize all potential locks for the set `C(v)`.

Since the `lock_held(t)` function requires a thread ID, we must also instrument a call to the POSIX `pthread_self()` function. This is handled by our `get_thread_id()` function. This instrumentation is performed per each function.

Rust

```rust
// instrument/src/race.rs

fn get_lock<'ctx>(instr: &InstructionValue<'ctx>) ->
Option<Lock<'ctx>> {
    if instr.get_opcode() != InstructionOpcode::Call ||
    instr.get_num_operands() != 2 {
        return None;
    }

    if let Some(operand) = instr.get_operand(1) {
        if let Some(func_name_val) = operand.left() {
            let func_name = cstr_to_str(func_name_val.get_name());

            let lock_typ = match func_name.as_str() {
                PTHREAD_MUTEX_LOCK => Some(LockTyp::Lock),
                PTHREAD_MUTEX_UNLOCK => Some(LockTyp::UnLock),
                _ => None,
            };
            if lock_typ.is_some() {
                let lock_operand = instr
                    .get_operand(0)
                    .unwrap()
                    .left()
                    .unwrap()
                    .into_pointer_value();
                return Some(Lock::new(lock_typ.unwrap(),
                lock_operand));
            }
        }
    }
    None
}

fn get_thread_id<'ctx>(
    context: &'ctx Context,
    module: &Module<'ctx>,
    builder: &Builder<'ctx>,
    func: &FunctionValue<'ctx>,
    instr: &InstructionValue<'ctx>,
    pthread_self_callsites: &mut HashMap<*mut LLVMValue,
    CallSiteValue<'ctx>>,
) -> Result<CallSiteValue<'ctx>> {
```

```rust
    let func_id = func.as_value_ref();
    let thread_id = match pthread_self_callsites.get(&func_id) {
        Some(thread_id) => *thread_id,
        None => {
            builder.position_before(&instr);
            let thread_id = build_pthread_self(context, module, builder)?;
            pthread_self_callsites.insert(func.as_value_ref(), thread_id);
            thread_id
        }
    };
    Ok(thread_id)
}
```

```rust
// instrument/src/inkwell_intrinsic.rs

fn get_pthread_self<'ctx>(context: &'ctx Context, module: &Module<'ctx>) -> FunctionValue<'ctx> {
    match get_func(module, PTHREAD_SELF) {
        Some(func) => func,
        None => {
            let pthread_self_type = context.i64_type().fn_type(&[], false);
            module.add_function(PTHREAD_SELF, pthread_self_type, None)
        }
    }
}

pub fn build_pthread_self<'ctx>(
    context: &'ctx Context,
    module: &Module<'ctx>,
    builder: &Builder<'ctx>,
) -> Result<CallSiteValue<'ctx>> {
    // instrument POSIX `pthread_self()` call
    let pthread_self_fn = get_pthread_self(context, module);
    let thread_id = builder.build_call(pthread_self_fn, &[], "")?;
    Ok(thread_id)
}
```

We insert `lock_held(t)` right after POSIX lock and unlock calls.

```rust
// instrument/src/inkwell_intrinsic.rs
```

```

fn get_update_lock_held<'ctx>(
    context: &'ctx Context,
    module: &Module<'ctx>,
) -> FunctionValue<'ctx> {
    match get_func(module, RACE_LOCK_HELD) {
        Some(func) => func,
        None => {
            let lock_held = context.void_type().fn_type(
                &[
                    context.i8_type().into(),
                    context.i64_type().into(),
                    context.i64_type().into(),
                ],
                false,
            );
            module.add_function(RACE_LOCK_HELD, lock_held, None)
        }
    }
}

pub fn build_update_lock_held<'ctx>(
    context: &'ctx Context,
    module: &Module<'ctx>,
    builder: &Builder<'ctx>,
    thread_id: CallSiteValue<'ctx>,
    lock: Lock<'ctx>,
) -> Result<()> {
    // instrument `__race_update_lock_held()`
    let lock_held = get_update_lock_held(context, module);
    builder.build_call(
        lock_held,
        &[
            context
                .i8_type()
                .const_int(lock.typ.i8() as u64, true)
                .into(),
            thread_id.try_as_basic_value().left().unwrap().into(),
            context
                .i64_type()
                .const_int(lock.lock.as_value_ref() as u64, true)
```

```
                .into(),
        ],
        "",
    )?;
    Ok(())
}
```

The instrumented function is named `__race_update_lock_held()`. It takes three parameters:

- Lock type: An integer indicating the operation (0 for unlock, 1 for lock).
- Thread ID: Thread identifier
- ID: A mutex identifier used for reporting purposes.

```LLVM-IR
%5 = call i64 @pthread_self()
...
%6 = call i32 @pthread_mutex_lock(ptr noundef @mutex1) #6, !dbg !81
call void @__race_update_lock_held(i8 1, i64 %5, i64 107913358915488),
...
%22 = call i32 @pthread_mutex_unlock(ptr noundef @mutex1) #6
call void @__race_update_lock_held(i8 0, i64 %5, i64 107913358915488)
...
```

In the following steps, we check if an instruction is a read or write operation. This is done by looking for load and store instructions. A load instruction indicates a read operation, and a store instruction indicates a write operation. Each of these instructions serves as a capture point to update `C(v)`.

This process is handled by the `handle_mem_access()` function.

```Rust
// instrument/src/race.rs

fn handle_mem_access<'ctx>(
    instr: &InstructionValue<'ctx>,
    operand: &BasicValueEnum<'ctx>,
    access_op: AccessOperation,
    global_int_vals: &HashSet<PointerValue<'ctx>>,
    lock_candidate_set: &mut HashMap<PointerValue<'ctx>, Vec<Lock<'ctx>>>,
    live_locks: &HashSet<Lock<'ctx>>,
    thread_id: CallSiteValue<'ctx>,
    context: &'ctx Context,
    module: &Module<'ctx>,
    builder: &Builder<'ctx>,
) -> Result<()> {
    if operand.is_pointer_value() {
```

```rust
        if let Some(global_var) = global_int_vals.get(&operand.into_pointer_value()) {
            lock_candidate_set.insert(
                operand.into_pointer_value(),
                live_locks.clone().into_iter().collect::<Vec<_>>(),
            );
            if let Some(next_instr) = instr.get_next_instruction() {
                builder.position_before(&next_instr);
            }
            let (line, _) = get_instr_loc(instr);
            build_update_shared_mem(
                context, module, builder, access_op, thread_id, global_var, line,
            )?;
        }
    }
    Ok(())
}
```

If a read or write operation does not involve a global variable, the function returns early without making any changes. If one is found, we append the `live_locks` to the initial `C(v)`. The implementation of this builder is as follows:

```rust
// instrument/src/inkwell_intrinsic.rs

fn get_updated_shared_mem<'ctx>(
    context: &'ctx Context,
    module: &Module<'ctx>,
) -> FunctionValue<'ctx> {
    match get_func(module, RACE_UPDATE_SHARED_MEM) {
        Some(func) => func,
        None => {
            let update_shared_mem = context.void_type().fn_type(
                &[
                    context.i8_type().into(),
                    context.i64_type().into(),
                    context.i64_type().into(),
                    context.i64_type().into(),
                ],
                false,
            );
            module.add_function(RACE_UPDATE_SHARED_MEM, update_shared_mem, None)
```

```
        }
    }
}

pub fn build_update_shared_mem<'ctx>(
    context: &'ctx Context,
    module: &Module<'ctx>,
    builder: &Builder<'ctx>,
    access_op: AccessOperation,
    thread_id: CallSiteValue<'ctx>,
    global_var: &PointerValue<'ctx>,
    line: u32,
) -> Result<()> {
    let update_shared_mem = get_updated_shared_mem(context, module);
    builder.build_call(
        update_shared_mem,
        &[
            context
                .i8_type()
                .const_int(access_op.i8() as u64, true)
                .into(),
            thread_id.try_as_basic_value().left().unwrap().into(),
            context
                .i64_type()
                .const_int(global_var.as_value_ref() as u64, true)
                .into(),
            context.i64_type().const_int(line as u64, true).into(),
        ],
        "",
    )?;
    Ok(())
}
```

There are four parameters:

1. The access operation kind, which is 0 for a load instruction and 1 for a store instruction.
2. The thread ID.
3. The global variable ID.
4. The line information where the instruction is located.

Here is the instrumented format.

LLVM-IR

```
%14 = load i32, ptr @counter, align 4
```

```
call void @__race_update_shared_mem(i8 0, i64 %5, i64 107913358889360, i64 14)
%15 = add i32 %14, 1
call void @__race_update_shared_mem(i8 1, i64 %5, i64 107913358889360, i64 14)
...
```

Lastly, the initial `C(v)` is initialized via the construction function. The builder implementation is described below.

```rust
// instrument/src/inkwell_intrinsic.rs

pub fn build_race_init<'ctx>(
    context: &'ctx Context,
    module: &Module<'ctx>,
    builder: &Builder<'ctx>,
    init_lock_candidate_set: &HashMap<PointerValue<'ctx>, Vec<Lock<'ctx>>>,
) -> Result<FunctionValue<'ctx>> {
    // __race_module_init() - initialize race module
    let init_func_typ = context.void_type().fn_type(&[], false);
    let init_func_typ_global_var = context.void_type().fn_type(
        &[
            // 1. global var name
            context.ptr_type(AddressSpace::default()).into(),
            // 2. global var decl line
            context.i64_type().into(),
            // 3. global var address
            context.i64_type().into(),
        ],
        false,
    );
    let init_func_typ_lock_var = context.void_type().fn_type(
        &[
            // 1. global var address
            context.i64_type().into(),
            // 2. lock var name
            context.ptr_type(AddressSpace::default()).into(),
            // 3. lock decl line
            context.i64_type().into(),
            // 4. lock address
            context.i64_type().into(),
        ],
```

```
        false,
    );

    let init_fn_global_var = module.add_function(
        RACE_INIT_CANDIDATE_LOCKSET_GLOBAL_VAR,
        init_func_typ_global_var,
        None,
    );
    let init_fn_lock_var = module.add_function(
        RACE_INIT_CANDIDATE_LOCKSET_LOCK_VAR,
        init_func_typ_lock_var,
        None,
    );

    // Create a constructor function to initialize race module
    let constructor = module.add_function(RACE_MODULE_INIT, init_func_typ, None);
    let entry = context.append_basic_block(constructor, RACE_INIT_ENTRY);
    builder.position_at_end(entry);

    for (global_var, locks) in init_lock_candidate_set {
        // initialize global variable
        let global_var_name = get_or_build_global_string_ptr(
            module,
            builder,
            &format!(
                "{}{}",
                RACE_GLOBAL_PREFIX,
                &cstr_to_str(global_var.get_name())
            ),
        )?;
        let (global_var_decl_line, _) = get_instr_loc(global_var);
        builder.build_call(
            init_fn_global_var,
            &[
                global_var_name.as_pointer_value().into(),
                convert_to_int_val(context, global_var_decl_line).into(),
                context
                    .i64_type()
```

```
                .const_int(global_var.as_value_ref() as u64, true)
                .into(),
        ],
        "",
    )?;

    // initialize lock variable
    for lock in locks {
        let lock_var_name = get_or_build_global_string_ptr(
            module,
            builder,
            &format!(
                "{}{}",
                RACE_GLOBAL_PREFIX,
                &cstr_to_str(lock.lock.get_name())
            ),
        )?;
        let (lock_var_decl_line, _) = get_instr_loc(&lock.lock);
        builder.build_call(
            init_fn_lock_var,
            &[
                context
                    .i64_type()
                    .const_int(global_var.as_value_ref() as u64, true)
                    .into(),
                lock_var_name.as_pointer_value().into(),
                convert_to_int_val(context, lock_var_decl_line).into(),
                context
                    .i64_type()
                    .const_int(lock.lock.as_value_ref() as u64, true)
                    .into(),
            ],
            "",
        )?;
    }
}
builder.build_return(None)?;
```

```
    Ok(constructor)
}
```

Global and mutex variables are the targets for instrumentation. Line information is also attached for reporting purposes. In the working example, we have two global variables `counter` and `n`. If a mutex is found (named, `mutex1`), the mutex variables also initialized for better reporting.

```llvm
define void @__race_module_init() {
__race_init_entry:
  call void @__race_init_candidate_lockset_global_var(ptr @__race.global.counter, i64 8, i64 107913358889360)
  call void @__race_init_candidate_lockset_lock_var(i64 107913358889360, ptr @__race.global.mutex1, i64 5, i64 107913358915488)
  call void @__race_init_candidate_lockset_global_var(ptr @__race.global.n, i64 9, i64 107913358913936)
  call void @__race_init_candidate_lockset_lock_var(i64 107913358913936, ptr @__race.global.mutex1, i64 5, i64 107913358915488)
  ret void
}
```

Now we're heading to the runtime handler implementation.

9.2.3.2 Runtime

The runtime exposes four functions as we instrumented in the module in the previous section.

```rust
// race_runtime/src/runtime.rs

#[no_mangle]
pub extern "C" fn __race_init_candidate_lockset_global_var(
    global_var_name: *const libc::c_char,
    global_var_decl_line: u32,
    global_var_id: i64,
) {
    init_candidate_lockset_global_var(global_var_name, global_var_decl_line, global_var_id);
}

#[no_mangle]
pub extern "C" fn __race_init_candidate_lockset_lock_var(
    global_var_id: i64,
    lock_var_name: *const libc::c_char,
    lock_var_decl_line: u32,
```

```
    lock_id: i64,
) {
    init_candidate_lockset_lock_var(global_var_id, lock_var_name, lock_var_decl_line, lock_id);
}

#[no_mangle]
pub extern "C" fn __race_update_lock_held(is_lock: i8, thread_id: i64, lock_id: i64) {
    update_lock_held(is_lock, thread_id, lock_id);
}

#[no_mangle]
pub extern "C" fn __race_update_shared_mem(
    is_write: i8,
    thread_id: i64,
    global_var_id: i64,
    line: i64,
) {
    update_shared_mem(thread_id, global_var_id);
    state_transition(is_write, thread_id, global_var_id, line);
}
```

The first two public functions initialize global variables (`C(v)`) and mutex variables. These are then managed through a singleton data structure.

```rust
// race_runtime/src/state.rs

fn cstr_to_string(ptr: *const libc::c_char) -> String {
    if ptr.is_null() {
        return "".to_string();
    }
    unsafe { std::ffi::CStr::from_ptr(ptr).to_string_lossy().into_owned() }
}

pub fn init_candidate_lockset_lock_var(
    global_var_id: i64,
    lock_var_name: *const libc::c_char,
    lock_var_decl_line: u32,
    lock_id: i64,
) {
    if lock_var_name.is_null() {
```

```
            return;
        }

        let lock_var_name = cstr_to_string(lock_var_name)
            .strip_prefix(RACE_GLOBAL_PREFIX)
            .unwrap()
            .to_string();
        lock_metadata.lock().unwrap().insert(
            lock_id,
            LockMetadata::new(lock_var_name, lock_var_decl_line),
        );
        lock_set
            .lock()
            .unwrap()
            .entry(global_var_id)
            .or_insert_with(BTreeSet::new)
            .insert(lock_id);

        init_lock_set
            .lock()
            .unwrap()
            .entry(global_var_id)
            .or_insert_with(BTreeSet::new)
            .insert(lock_id);
    }

    pub fn init_candidate_lockset_global_var(
        global_var_name: *const libc::c_char,
        global_var_decl_line: u32,
        global_var_id: i64,
    ) {
        if global_var_name.is_null() {
            return;
        }

        let global_var_name = cstr_to_string(global_var_name)
            .strip_prefix(RACE_GLOBAL_PREFIX)
            .unwrap()
            .to_string();
        global_var_metadata.lock().unwrap().insert(
            global_var_id,
```

```
        GlobalVarMetadata::new(global_var_name, global_var_decl_line),
    );
    // set init state
    // as we only consider global variables for target shared memory,
    // directly set `Virgin` state initially
    init_state(global_var_id);
}
```

```rust
// race_runtime/src/state.rs

const RACE_TEST_ENABLED: &str = "RACE_UNIT_TEST_ENABLED";

pub struct GlobalVarMetadata {
    global_var_name: String,
    global_var_decl_line: u32,
}
impl GlobalVarMetadata {
    fn new(global_var_name: String, global_var_decl_line: u32) -> Self {
        Self {
            global_var_name,
            global_var_decl_line,
        }
    }
}

pub struct LockMetadata {
    lock_var_name: String,
    lock_var_decl_line: u32,
}
impl LockMetadata {
    fn new(lock_var_name: String, lock_var_decl_line: u32) -> Self {
        Self {
            lock_var_name,
            lock_var_decl_line,
        }
    }
}

#[derive(Eq, Hash, PartialEq)]
```

```
pub struct Reported {
    thread_id: i64,
    global_var_id: i64,
    global_var_decl: u32,
    global_var_used: i64,
}
impl Reported {
    fn new(thread_id: i64, global_var_id: i64, global_var_decl: u32,
    global_var_used: i64) -> Self {
        Self {
            thread_id,
            global_var_id,
            global_var_decl,
            global_var_used,
        }
    }
}

#[derive(Debug, PartialEq)]
pub enum State {
    Virgin,
    Exclusive,
    Shared,
    SharedModified,
}

pub struct ThreadState {
    thread_ids: HashSet<i64>,
    state: State,
}
impl Default for ThreadState {
    fn default() -> Self {
        Self {
            thread_ids: HashSet::new(),
            state: State::Virgin,
        }
    }
}

lazy_static::lazy_static! {
    // <global var id, metadata>
```

```
        pub static ref global_var_metadata: Arc<Mutex<HashMap<i64, GlobalVarMetadata>>> = Arc::new(Mutex::new(HashMap::new()));
        // <lock  id, metadata>
        pub static ref lock_metadata: Arc<Mutex<HashMap<i64, LockMetadata>>> = Arc::new(Mutex::new(HashMap::new()));
        // <global var id, lockset>
        pub static ref init_lock_set: Arc<Mutex<HashMap<i64, BTreeSet<i64>>>> = Arc::new(Mutex::new(HashMap::new()));
        pub static ref lock_set: Arc<Mutex<HashMap<i64, BTreeSet<i64>>>> = Arc::new(Mutex::new(HashMap::new()));
        // <thread id, lockset>
        pub static ref lock_held: Arc<Mutex<HashMap<i64, BTreeSet<i64>>>> = Arc::new(Mutex::new(HashMap::new()));
        // <global var id, state>
        pub static ref state: Arc<Mutex<HashMap<i64, ThreadState>>> = Arc::new(Mutex::new(HashMap::new()));
        // <set>
        pub static ref reported: Arc<Mutex<HashSet<Reported>>> = Arc::new(Mutex::new(HashSet::new()));
}
```

The most important data structure is `ThreadState`, which holds the current state from the four possible states and a set of thread IDs.

```rust
// race_runtime/src/state.rs

fn init_state(global_var_id: i64) {
    state
        .lock()
        .unwrap()
        .insert(global_var_id, ThreadState::default());
}
```

When a global variable is initialized, its state is also initialized to the starting point of the state-transition diagram, which is the `Virgin` state.

The handler of `lock_held(t)` is implemented as below.

```rust
// race_runtime/src/state.rs

pub fn update_lock_held(is_lock: i8, thread_id: i64, lock_id: i64) {
    if let Some(lock_typ) = LockTyp::from_i8(is_lock) {
        match lock_typ {
            LockTyp::Lock => {
                lock_held
                    .lock()
                    .unwrap()
```

```
                    .entry(thread_id)
                    .or_insert_with(BTreeSet::new)
                    .insert(lock_id);
            }
            LockTyp::UnLock => {
                if let Some(set) = lock_held.lock().unwrap().get_mut(&thread_id) {
                    set.remove(&lock_id);
                }
            }
        }
    }
}
```

The lock_held data structure is a map that uses thread IDs as keys and a list of lock IDs as values. When an unlock operation is performed, the corresponding lock ID is removed from the list.

The __race_update_shared_mem function manages both the refinement of C(v) and the state transitions. Within its body, it calls update_shared_mem() to refine C(v) and state_transition() to handle the state changes.

```rust
// race_runtime/src/state.rs

pub fn update_shared_mem(thread_id: i64, global_var_id: i64) {
    // calc. C(v) = C(v) ∩ lock_held(t)
    match lock_held.lock().unwrap().get(&thread_id) {
        Some(holds) => {
            let mut new_lockset = BTreeSet::new();
            if let Some(set) = lock_set.lock().unwrap().get(&global_var_id) {
                for l in set {
                    if holds.contains(l) {
                        new_lockset.insert(*l);
                    }
                }
            }
            lock_set.lock().unwrap().insert(global_var_id, new_lockset);
        }
        None => {
            lock_set
                .lock()
                .unwrap()
                .insert(global_var_id, BTreeSet::new());
```

```
        }
    }
}
```

At each `load` and `store` instruction is met, the handler is called, and `C(v)` is refined.

```rust
// race_runtime/src/state.rs

pub fn state_transition(is_write: i8, thread_id: i64, global_var_id: i64, line: i64) {
    if let Some(access_op) = AccessOperation::from_i8(is_write) {
        let mut s = state.lock().unwrap();
        match access_op {
            AccessOperation::Write => {
                if let Some(cur_state) = s.get_mut(&global_var_id) {
                    match cur_state.state {
                        State::Virgin => {
                            cur_state.state = State::Exclusive;
                            cur_state.thread_ids.insert(thread_id);
                        }
                        State::Exclusive => {
                            if !cur_state.thread_ids.contains(&thread_id) {
                                cur_state.state = State::SharedModified;
                                cur_state.thread_ids.insert(thread_id
                            }
                        }
                        State::Shared => {
                            cur_state.state = State::SharedModified;
                            cur_state.thread_ids.insert(thread_id);
                        }
                        State::SharedModified => {
                            if let Some(set) = lock_set.lock().unwrap().get(&global_var_
                            {
                                if set.is_empty() {
                                    report(thread_id, global_var_id, line);
                                }
                            }
                        }
```

```
                    }
                }
            }
            AccessOperation::Read => {
                if let Some(cur_state) = s.get_mut(&global_var_id) {
                    if cur_state.state == State::Exclusive {
                        if !cur_state.thread_ids.contains(&thread_id) {
                            cur_state.state = State::Shared;
                            cur_state.thread_ids.insert(thread_id);
                        }
                    }
                }
            }
        }
    }
}
```

For each of the four states, the conditions and corresponding state changes are implemented in a match arm. A data race is only reported in the `SharedModified` state. The reporting function is implemented as follows:

```rust
// race_runtime/src/state.rs

pub fn report(thread_id: i64, global_var_id: i64, line: i64) {
    let gv_md = global_var_metadata.lock().unwrap();
    let md = gv_md.get(&global_var_id).unwrap();
    let report = Reported::new(thread_id, global_var_id, md.global_var_decl_line, line);

    let mut r = reported.lock().unwrap();
    if r.get(&report).is_none() {
        println!(
            "{}",
            format!(
                "[--------------------- Data race detected #{} ---------------------]",
                r.len()
            )
            .red()
            .bold()
        );
        if !is_test_enabled() {
```

```
            println!("thread id          = {}", thread_id);
        }
        println!("variable name      = {}", md.global_var_name);
        println!("variable decl      = {}", md.global_var_decl_line);
        println!("variable used line = {}", line);
        println!("[related locks]");

        if let Some(set) = init_lock_set.lock().unwrap().get(&global_var_id) {
            for lock_id in set {
                let lk_md = lock_metadata.lock().unwrap();
                let md = lk_md.get(lock_id).unwrap();
                println!("    - lock variable name = {}", md.lock_var_name);
                println!("    - lock variable decl = {}", md.lock_var_decl_line);
            }
        }
        println!("");
        r.insert(report);
    }
}

fn is_test_enabled() -> bool {
    matches!(env::var(RACE_TEST_ENABLED), Ok(val) if val == "1")
}
```

The report provides helpful information such as the variable's name, its declaration and usage locations, and any related mutex variables.

Now, let's run the initial data race example. The code is attached below.

```c
// example-code/race/race-example-fix.c

#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

volatile int counter = 0;
int n = 20000;

void* increment_counter(void* arg) {
    for (int i = 0; i < n; i++) {
        counter++;
```

```c
    }
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    if (pthread_create(&thread1, NULL, increment_counter, NULL) != 0) {
        perror("Error creating thread 1");
        return 1;
    }
    if (pthread_create(&thread2, NULL, increment_counter, NULL) != 0) {
        perror("Error creating thread 2");
        return 1;
    }
    if (pthread_join(thread1, NULL) != 0) {
        perror("Error joining thread 1");
        return 1;
    }
    if (pthread_join(thread2, NULL) != 0) {
        perror("Error joining thread 2");
        return 1;
    }

    printf("Expected counter value: %d\n", 2 * n);
    printf("Actual counter value: %d\n", counter);

    return 0;
}
```

Shell

```shell
> clang -g -O0 -S -emit-llvm race-example.c -o race-example.ll
...
> ./instrument -i race-example.ll  -o irace-example.ll -m race
...
> clang irace-example.ll  -L. -lrace_runtime
...
> LD_LIBRARY_PATH=./ ./a.out
❯ LD_LIBRARY_PATH=./ ./a.out
[-------------------- Data race detected #0 --------------------]
```

```
thread id          = 136849857177280
variable name      = counter
variable decl      = 5
variable used line = 10
[related locks]

[-------------------- Data race detected #1 --------------------]
thread id          = 136849865569984
variable name      = counter
variable decl      = 5
variable used line = 10
[related locks]

Expected counter value: 40000
Actual counter value: 27338
```

Let's fix the example program and confirm that our detector no longer reports a data race.

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

pthread_mutex_t mutex;
volatile int counter = 0;
int n = 20000;

void* increment_counter(void* arg) {
    pthread_mutex_lock(&mutex);
    for (int i = 0; i < n; i++) {
        counter++;
    }
    pthread_mutex_unlock(&mutex);
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    if (pthread_create(&thread1, NULL, increment_counter, NULL) != 0) {
        perror("Error creating thread 1");
        return 1;
```

```c
    }
    if (pthread_create(&thread2, NULL, increment_counter, NULL) != 0) {
        perror("Error creating thread 2");
        return 1;
    }
    if (pthread_join(thread1, NULL) != 0) {
        perror("Error joining thread 1");
        return 1;
    }
    if (pthread_join(thread2, NULL) != 0) {
        perror("Error joining thread 2");
        return 1;
    }

    printf("Expected counter value: %d\n", 2 * n);
    printf("Actual counter value: %d\n", counter);

    return 0;
}
```

The compile and run command is as follows:

```shell
> clang -g -O0 -S -emit-llvm race-example-fix.c -o race-example-fix.ll
...
> ./instrument -i race-example-fix.ll -o irace-example-fix.ll -m race
...
> clang irace-example-fix.ll  -L. -lrace_runtime
...
> LD_LIBRARY_PATH=./ ./a.out
Expected counter value: 40000
Actual counter value: 40000
```

A misuse of a mutex can lead to a data race. This is the last demonstration, and our detector should report it.

```rust
// example-code/race/race-example-misuse-mutex.c

#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

pthread_mutex_t mutex1;
```

```
pthread_mutex_t mutex2;
volatile int counter = 0;
int n = 20000;

void* increment_counter1(void* arg) {
    pthread_mutex_lock(&mutex1);
    for (int i = 0; i < n; i++) {
        counter++;
    }
    pthread_mutex_unlock(&mutex1);
    return NULL;
}

void* increment_counter2(void* arg) {
    pthread_mutex_lock(&mutex2);
    for (int i = 0; i < n; i++) {
        counter++;
    }
    pthread_mutex_unlock(&mutex2);
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    if (pthread_create(&thread1, NULL, increment_counter1, NULL) !=
    0) {
        perror("Error creating thread 1");
        return 1;
    }
    if (pthread_create(&thread2, NULL, increment_counter2, NULL) !=
    0) {
        perror("Error creating thread 2");
        return 1;
    }
    if (pthread_join(thread1, NULL) != 0) {
        perror("Error joining thread 1");
        return 1;
    }
    if (pthread_join(thread2, NULL) != 0) {
        perror("Error joining thread 2");
        return 1;
```

```c
    }

    printf("Expected counter value: %d\n", 2 * n);
    printf("Actual counter value: %d\n", counter);

    return 0;
}
```

Our detector reports it as below:

```shell
[-------------------- Data race detected #0 --------------------]
thread id          = 131363739002560
variable name      = counter
variable decl      = 7
variable used line = 22
[related locks]
    - lock variable name = mutex1
    - lock variable decl = 5
    - lock variable name = mutex2
    - lock variable decl = 6

[-------------------- Data race detected #1 --------------------]
thread id          = 131363747395264
variable name      = counter
variable decl      = 7
variable used line = 13
[related locks]
    - lock variable name = mutex1
    - lock variable decl = 5
    - lock variable name = mutex2
    - lock variable decl = 6

Expected counter value: 40000
Actual counter value: 25445
```

We have successfully implemented and demonstrated a practical data race detector. There are several areas for further improvement, which are left as homework for continued study.

9.2.4 Homework

Exercise 9.1 We suggest the following topics for further study and exploration:

- **Topic #1: Extend the Race detactor to capture dynamic(heap) data**: Currently, our implementation only captures static global variables. Heap data, however, can be initialized in one function and persist across other function scopes. Readers can extend the program to address this limitation.

Part IX

Appendix

Next Journey

λ

10. Conclusion

10.1 Further Study

This book originated from my personal study. While I have some experience with static analysis, I had little practical experience in implementing dynamic analysis. My initial understanding was based on what I had heard about the problems and solutions in this field from coworkers.

The field of PL is both challenging and fascinating, with a vast scope. To guide your further exploration, I have outlined some key topics for continued study and research.

- **Compiler**: I agree that compilers are one of the most important components in computer science. Their applications are incredibly vast.

  For example, the demand for new hardware to achieve high performance in AI systems is growing rapidly. Compilers must be able to generate efficient code for these target machines, which now include specialized hardware like ASICs. This has created a need for developing new compiler backends.

  Furthermore, many concepts have been developed to generate faster code, such as operator fusion, code reordering (scheduling), and equality saturation. In the context of program analysis and engineering, a deep understanding of compiler passes—and the ability to modify them—is a valuable skill.

  Some projects develop an IR from scratch for their optimization passes. To better meet the needs of new domains, multi-level IR is a popular concept for enabling more effective optimization. LLVM, for instance, allows programmers to define and generate their own custom IR and apply fine-grained, layer-specific optimizations.
- **Type System**: The type system is a fascinating and challenging area of computer science. In 2025, most programmers would agree on its importance. The success of languages like TypeScript and the growing popularity of Rust are testaments to this. A strong type system's main contribution is providing a safe programming environment. By catching type-related bugs at compile time, it eliminates a significant class of runtime errors that are often difficult to debug.

  Type inference, a concept developed from type theory, allows a compiler to determine a variable's type without explicit annotation.

  In new domains, such as AI compilers, there's a growing demand to design new type systems that can efficiently handle complex mathematical calculations.

- **Program Analysis**: Program bugs are ubiquitous. There are two primary approaches to finding them: static analysis and dynamic analysis. In this book, we have explored the dynamic-analysis approach, but static analysis is also widely adopted.

  The key difference lies in their scope and guarantees. Static analysis, in theory, can find all bugs within a defined domain, but it often involves a trade-off between time constraints and accuracy. Achieving perfect accuracy in a reasonable amount of time is not theoretically guaranteed.

  In contrast, if a bug is detected by a dynamic analysis tool, it is a real, confirmed bug. However, dynamic analysis is limited in its scope because it is impossible to explore all possible program states at runtime. This is why hybrid analysis, which combines both approaches, has been so successful in academia—it amortizes the weaknesses of each method.

  Developing a strong understanding of both static and dynamic analysis is crucial. This combined knowledge provides a more robust toolkit for tackling new problems and designing effective solutions.
- **Verification**: Even with the best software hardening efforts from program analyzers, programs can still contain unknown bugs. So, how can we guarantee that a program is bug-free?

  In the real world, we use unit tests to check a function's behavior across a few scenarios and inputs. For example, if you implement an `add(a, b)` function, you might write tests like `add(1, 2)` or `add(100, 200)`. However, no matter how many test cases you write, you can never achieve a theoretical guarantee of correctness.

  This is where formal verification comes in. It guarantees correctness by defining theorems and lemmas about an operation, such as addition, and then formally verifying that the implementation meets those definitions. Once verified, it can generate certified code in a high-level language.

  While formal verification is costly, difficult to learn, and challenging to apply broadly, it is a powerful tool for ensuring correctness in specific, mission-critical domains. It's widely used in areas like compilers, programming languages, aircraft control systems, and other applications where correctness is paramount.
- **Hardware Abstraction**: Even though hardware design and development aren't directly correlated with programming languages, the area is promising and interesting. Here are a few minor points to consider:

  Abstraction is crucial in many fields, including programming language design and microarchitecture. However, the maturity of hardware abstraction lags behind the rapid growth of software. A robust, end-to-end abstraction—covering the processor pipeline, ISA, and communication with co-processors and drivers—would make the architecture more transparent and intuitive for software developers who interact with it, such as those working on compilers and SDKs.
- **AI-based ?**: Every day, a lot of news in the AI field is announced, and the results seem incredible. Sometimes I imagine developing an AI-based drone that could perform interesting tasks like delivery, observation, or running errands. These tasks are tailored for an AI-based approach.

  Likewise, there are many open things to investigate and develop. Even if they're not used for software development, there are so many fun things in the world. It's great to imagine my own original ideas on how to use AI or to learn about others' ideas from papers.

This book began as a personal journey of study, so it may contain logical or implementation errors. I kindly ask for your patience and would be grateful for any corrections you may find. Your feedback is invaluable.

If you find this book to be a helpful resource, I would be deeply appreciative if you would recommend it to your friends and coworkers. Thank you so much for taking the time to read this book.

Sincerely,

Hyunsoo Shin

Bibliography

[1] Wikipedia, “Language-based security.”
[2] “Clang Compiler,” [Online]. Available: https://clang.llvm.org/
[3] “LLVM Pass Tutorial,” [Online]. Available: https://llvm.org/docs/WritingAnLLVMNewPMPass.html
[4] “POSIX,” [Online]. Available: https://en.wikipedia.org/wiki/POSIX
[5] “Reflection Programming,” [Online]. Available: https://en.wikipedia.org/wiki/Reflective_programming
[6] “Single Static Assignment,” [Online]. Available: https://en.wikipedia.org/wiki/Static_single-assignment_form
[7] “Code Coverage,” [Online]. Available: https://en.wikipedia.org/wiki/Code_coverage
[8] “Hardhat Coverage,” [Online]. Available: https://hardhat.org/hardhat-runner/docs/guides/test-contracts
[9] “Inkwell,” [Online]. Available: https://github.com/TheDan64/inkwell
[10] “Construction function in C,” [Online]. Available: https://wiki.osdev.org/Calling_Global_Constructors
[11] “atexit,” [Online]. Available: https://man7.org/linux/man-pages/man3/atexit.3.html
[12] “Mangling,” [Online]. Available: https://en.wikipedia.org/wiki/Name_mangling
[13] “Dominance Graph,” [Online]. Available: https://en.wikipedia.org/wiki/Dominator_(graph_theory)
[14] “Undefined Behavior,” [Online]. Available: https://en.wikipedia.org/wiki/Undefined_behavior
[15] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “AddressSanitizer: A Fast Address Sanity Checker,” 2012, [Online]. Available: https://www.usenix.org/system/files/conference/atc12/atc12-final39.pdf
[16] “Uninitialized Read,” [Online]. Available: https://en.wikipedia.org/wiki/Uninitialized_variable
[17] “Buffer Overflow,” [Online]. Available: https://en.wikipedia.org/wiki/Buffer_overflow
[18] “Double Free,” [Online]. Available: https://en.wikipedia.org/wiki/Memory_safety
[19] “Null Dereference,” [Online]. Available: https://en.wikipedia.org/wiki/Null_pointer
[20] “Clang ASAN,” [Online]. Available: https://clang.llvm.org/docs/AddressSanitizer.html
[21] “Fuzzing,” [Online]. Available: https://en.wikipedia.org/wiki/Fuzzing
[22] “AFL,” [Online]. Available: https://github.com/google/AFL/blob/61037103ae3722c8060ff7082994836a794f978e/docs/technical_details.txt
[23] “mmap,” [Online]. Available: https://man7.org/linux/man-pages/man2/mmap.2.html
[24] “eventfd,” [Online]. Available: https://man7.org/linux/man-pages/man2/eventfd.2.html

[25] "concolic testing," [Online]. Available: https://en.wikipedia.org/wiki/Concolic_testing
[26] "z3," [Online]. Available: https://github.com/Z3Prover/z3
[27] "Maze Solver," [Online]. Available: https://feliam.wordpress.com/2010/10/07/the-symbolic-maze/
[28] "Delta Debugging Wiki," [Online]. Available: https://en.wikipedia.org/wiki/Delta_debugging
[29] "Delta Debugging Paper," [Online]. Available: https://www.cs.purdue.edu/homes/xyzhang/fall07/Papers/delta-debugging.pdf
[30] "Large Language Model," [Online]. Available: https://en.wikipedia.org/wiki/Large_language_model
[31] "Convolutional Neural Network," [Online]. Available: https://en.wikipedia.org/wiki/Convolutional_neural_network
[32] "MNIST Dataset," [Online]. Available: https://en.wikipedia.org/wiki/MNIST_database
[33] "Natural Language Processing," [Online]. Available: https://en.wikipedia.org/wiki/Natural_language_processing
[34] "Long short-term memory," [Online]. Available: https://en.wikipedia.org/wiki/Long_short-term_memory
[35] C. S. Xia, M. Paltenghi, J. L. Tian, M. Pradel, and L. Zhang, "Fuzz4All: Universal Fuzzing with Large Language Models," 2024, [Online]. Available: https://arxiv.org/pdf/2308.04748
[36] "Fuzz4all Bug Finding," [Online]. Available: https://github.com/golang/go/issues/61158
[37] "Program Synthesis," [Online]. Available: https://en.wikipedia.org/wiki/Program_synthesis
[38] "Qwen Model," [Online]. Available: https://huggingface.co/Qwen/Qwen2.5-Coder-7B-Instruct
[39] "Flash Fill," [Online]. Available: https://support.microsoft.com/en-us/office/using-flash-fill-in-excel-3f9bcf1e-db93-4890-94a0-1578341f73f7
[40] "Volatile Keyword," [Online]. Available: https://en.wikipedia.org/wiki/Volatile_(computer_programming)
[41] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. AndersonAuthors, "Eraser: a dynamic data race detector for multithreaded programs," 1997, [Online]. Available: https://dl.acm.org/doi/10.1145/265924.265927
[42] L. Lamport, "Time, clocks, and the ordering of events in a distributed system ," 1978, [Online]. Available: https://dl.acm.org/doi/10.1145/359545.359563