

Rust-based Path Coverage-Guided Fuzzing

by

Yunji Kim

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2024

© Yunji Kim 2024

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Coverage-guided fuzzing is one of the most effective approaches for library testing.[1] While edge coverage has proven successful in finding many bugs, security-critical projects often require higher granularity to thoroughly examine complex execution paths.[2][3] Path coverage offers a promising alternative, but it is hindered by path explosion and the overhead of path handling.

In this thesis, we propose Bounded Path coverage, an advanced coverage metric that mitigates path explosion by leveraging a configurable loop unrolling parameter. For that we propose two algorithms: DAGification and Path reduction. To balance thorough path exploration with resource efficiency, we use the Rust compiler toolchain’s MIRI[4][5] component with minimal instrumentation overhead for both static and runtime analyses. Our prototype fuzzer successfully generated bounded path coverage, uncovered one unknown bug and one discrepancy from real-world Rust projects, and showcase the potential of superior path exploration compared to traditional edge coverage.

Acknowledgements

I would like to thank my supervisor, Professor Meng Xu, for his expert guidance on algorithm design and Rust programming during this work. I am grateful for his incredible patience and insight. I thank my thesis readers Chengnian Sun and Sihang Liu for their valuable suggestions that helped make this thesis. I would also like to thank my colleagues and the CrySP Group for their encouraging words during my time here.

Table of Contents

Author’s Declaration	ii
Abstract	iii
Acknowledgements	iv
List of Figures	viii
List of Tables	ix
1 Introduction	1
2 Related Work	3
2.1 Coverage-based Fuzz Testing	3
2.2 Fuzzing Tools for Rust	5
3 Background	6
3.1 RUST language and MIR Interpreter	6
3.2 LLVM Loop Terminology	7
4 Approach	9
4.1 Overall Framework	9
4.2 DAGification Algorithm	11

4.2.1	Kosaraju’s Algorithm	12
4.2.2	LLVM LoopSimplify	12
4.2.3	Decomposing Loops	12
4.3	Path Reduction Algorithm	13
4.3.1	Case 1 (Entering Edge): $k < 0$	13
4.3.2	Case 2: $k = 0$	14
4.3.3	Case 3 (Exiting Edge): $k > 0$	15
5	Implementation	16
5.1	Static Analysis	16
5.1.1	Static CFG Dump	17
5.1.2	DAGification	17
5.2	Runtime Analysis	19
5.2.1	Runtime Trace Dump	19
5.2.2	Path Reduction	19
6	Evaluation	23
6.1	Evaluation Setup	23
6.1.1	Dataset	23
6.1.2	Test Oracle	24
6.1.3	Experiment Infrastructure	24
6.2	Bug-Finding Ability (RQ1)	24
6.2.1	Case Study 1: Move Bytecode Verifier	25
6.2.2	Case Study 2: Real-World Bug Discovery	25
6.3	Validity of Loop Unrolling Parameter (RQ2)	26
6.4	Coverage Growth Comparison (RQ3)	29
7	Future Work	32

8 Conclusion	34
References	35

List of Figures

4.1 Overall Framework	10
5.1 DAGification	17
5.2 DAGification-Recursive	18
5.3 Path Reduction Flowchart	21
6.1 RQ2 time	27
6.2 RQ2 fixedbitset	28
6.3 RQ3 fixedbitset	30

List of Tables

6.1	Summary of the dataset	24
6.2	Bounded path coverage size for each parameter	26

Chapter 1

Introduction

Rust is a rapidly growing programming language, particularly in the security field. The compiler and bytecode verifier of Move, a blockchain language, are written in Rust. Despite the critical importance of security testing in some Rust projects, testing for rust still relies on traditional methodologies such as unit testing, manual code review, or wrappers of C/C++ fuzzing tools. It is surprising to find that no fuzzing tools have been specifically developed in Rust for security-high-demand Rust software.

In this thesis, I propose a Rust-based path-coverage guided fuzzer, which is designed to rigorously test certain targets or components that require thorough examination using more granular coverage metrics. I aim to demonstrate that it is feasible to test real-world projects using a fuzzer that leverages path coverage as feedback, and that this approach increases the potential for a more thorough examination of the project. I evaluate this using my own prototype fuzzer, written in Rust, targeting Rust crate projects to demonstrate its feasibility and effectiveness.

The key limitation of current state-of-the-art coverage metrics, edge coverage, can be summarized as superficial path exploration. They often fail to explore deeply nested or rare paths that may contain critical vulnerabilities. On the other hand, path coverage also has limitation, which is path explosion and overuse of resources. Path coverage, while comprehensive, is impractical due to the exponential growth in the number of possible paths as the software size and code complexity increases. This results in fuzzers focusing on code regions with simpler control flow, making it infeasible for path coverage based fuzzers to exhaustively explore all paths.

These limitations point to a need for more advanced coverage metrics that can strike a balance between the comprehensiveness of path coverage and the feasibility of lighter-

weight metrics like block or edge coverage. This gap motivates the introduction of Bounded Path Coverage, a novel metric that aims to explore a bounded subset of paths based on configurable parameter, unrolling depth. The proposed metric seeks to address the shortcomings of current fuzzing techniques by directing fuzzers toward deeper, more meaningful execution paths where bugs are more likely to reside, without incurring the prohibitive overhead of full path coverage.

Chapter 2

Related Work

2.1 Coverage-based Fuzz Testing

Coverage-guided fuzz testing is a powerful technique widely used in software testing to discover vulnerabilities and bugs. [6] It operates by generating a large number of random inputs for the software under test and then monitoring code execution to assess how much of the code is exercised. The primary aim of this fuzzer is to maximize the variety of code paths explored by the fuzzer, which increases the likelihood of revealing bugs, especially those located in rarely executed paths. Code coverage metrics are central to this process, serving as a guide to evaluate the effectiveness of a fuzzer in exploring the codebase.

Several code coverage metrics are used to measure the extent to which the code under test has been exercised by a set of inputs. Some of the most common coverage metrics include Function Coverage, Statement/Line Coverage, Block Coverage, Branch/Edge Coverage and Path Coverage. While these traditional metrics have proven useful, they have limitations in complex systems where bugs often hide in deep, hard-to-reach code paths.

In modern fuzz testing, edge coverage has emerged as a dominant metric due to its balance between simplicity/granularity and effectiveness. Edge coverage tracks whether all control flow transitions between basic blocks have been executed. It offers more nuanced insight than statement or block coverage but remains computationally feasible compared to path coverage. State-of-the-art fuzzers, such as American Fuzzy Lop (AFL) [7], LibFuzzer [8], and Honggfuzz [9], leverage edge coverage to guide their exploration.

AFL [7] is a widely used modern fuzzing tool that instruments target binaries at compile-time using compilers such as GCC or Clang. It generates inputs aimed at maximizing the discovery of new code edges, leveraging a coverage-guided strategy. LibFuzzer [8], on

the other hand, operates as an in-process fuzzing engine integrated into the LLVM/Clang toolchain. By running directly within the target program’s memory space, LibFuzzer offers improved performance and seamless integration with sanitizers, making it effective for development-stage fuzzing. Honggfuzz [9] builds upon some of AFL’s features, adding advanced capabilities such as dynamic instrumentation and optional taint analysis. Despite its improvements, Honggfuzz, like AFL, relies primarily on edge coverage, limiting its ability to navigate intricate program behaviors and uncover vulnerabilities hidden behind complex control structures.

To address the limitations of edge coverage in exploring complex program behaviors, several studies have explored enhancements to coverage metrics. VUzzer [2], for instance, employs a lightweight dynamic analysis to prioritize inputs based on control-flow transitions and execution paths, improving the exploration of hard-to-reach code. However, it does not fully realize path coverage as a standalone metric but instead uses heuristics to approximate deeper path exploration. Similarly, PATA [10] extends traditional edge coverage by incorporating path-sensitive feedback through abstract execution path trees, enabling it to handle some complex control-flow scenarios. PathAFL [11] takes a unique approach by introducing the concept of an "h-path", defined as a new path where all edges have been previously explored. PathAFL utilizes selective instrumentation and efficient filtering algorithms to identify high-weight h-paths, add them to the seed queue, and prioritize them during fuzzing. This results in better seed selection and power scheduling, enabling PathAFL to achieve significant improvements in path exploration and bug discovery compared to AFL and other advanced fuzzers.

VUzzer, PATA, and PathAFL collectively illustrate efforts to extend the traditional edge coverage paradigm by incorporating varying degrees of path-level feedback [2, 10, 11]. While these approaches enhance fuzzing efficiency and bug discovery, they do not fully exploit path coverage as a distinct metric capable of identifying precise sequences of execution paths. They can rather be interpreted as an augmented form of edge coverage that provides additional context about program branches and execution paths. [3] For instance, PAFL achieves notable improvements, yet it primarily optimizes within the edge-coverage framework. This gap underscores the need for fuzzing techniques that utilize true path coverage as a core feedback mechanism. Such an approach could offer a more comprehensive exploration of program states, particularly in languages like Rust, where complex control-flow patterns and unique safety guarantees demand sophisticated coverage strategies.

2.2 Fuzzing Tools for Rust

Existing fuzzers for Rust projects primarily involve well-known C/C++-targeted fuzzers, such as AFL and LibFuzzer. [12, 13, 14] Since Rust’s compilation process and toolchain differ from C/C++’s, for AFL and LibFuzzer to work, the target program must be compiled with special instrumentation to collect coverage feedback. `af1.rs` [12] is a third-party wrapper, which allows to run on code written in the Rust language by providing bindings to integrate AFL with Rust. Also, `cargo-fuzz` [13] is itself not a fuzzer, but a tool to invoke libFuzzer. `cargo-fuzz` is generally a better and easier choice for fuzzing Rust code since it is tailored specifically for Rust, which is integrated into the Rust ecosystem more seamlessly.

Using AFL or LibFuzzer via wrappers for Rust is not as straightforward or optimized as using tools designed specifically for Rust. To improve the fuzzing performance of wrapper tools targeting Rust programs, one of the most explored approaches has been the enhancement of fuzz target generation. There has been work on creating fuzz targets specifically tailored for Rust, such as RULF [15] and RPG [16]. The fuzz targets generated by both RULF and RPG were evaluated using AFL.

RULF (*Rust Language Fuzzer*) [15] focuses on the automatic generation of fuzz targets by analyzing Rust code and identifying key entry points suitable for fuzzing. It leverages static and dynamic analysis to produce efficient fuzz targets that maximize code coverage. On the other hand, RPG (*Rust Program Generator*) [16] takes a complementary approach by generating entire Rust programs that adhere to the language’s syntax and semantics. These generated programs are then used as inputs to evaluate and guide the fuzzing process. Both tools aim to bridge the gap between Rust’s unique programming paradigms and the requirements of traditional fuzzing frameworks, ultimately enhancing fuzzing efficiency and effectiveness for Rust applications.

Chapter 3

Background

3.1 RUST language and MIR Interpreter

Rust is a programming language designed with a strong emphasis on memory safety, performance, and concurrency, all while maintaining control over low-level details. These characteristics make Rust an ideal choice for projects that require high levels of security, particularly in fields such as compiler construction, blockchain, and cryptography. A notable example is the Move Compiler, which is utilized in blockchain ecosystems like Aptos and Diem (formerly Libra).

The Rust compilation process consists of several stages, one of which is the generation of the Mid-level Intermediate Representation (MIR). MIR is a simplified, structured representation of a Rust program that abstracts away high-level syntax while retaining critical details about control flow and data flow. Positioned between the high-level abstract syntax tree (AST) and the low-level LLVM intermediate representation (IR), MIR serves as a pivotal layer in the Rust compilation pipeline, enabling advanced analysis and optimization.

A key tool within the Rust ecosystem is **MIRI** [4] [5], an interpreter specifically designed to execute Rust programs at the MIR level. MIRI provides a precise and deterministic execution environment, allowing developers to simulate and analyze Rust programs without requiring full compilation to machine code. This capability is particularly beneficial for early-stage bug detection, the identification of undefined behaviors and verifying and interpreting ‘unsafe’ code.

MIRI’s ability to efficiently analyze and interpret execution paths creates a powerful foundation for this research. By leveraging MIRI, the proposed fuzzing framework can

achieve detailed path coverage with minimal performance impact (instrumentation overhead), focusing on uncovering security vulnerabilities embedded in complex control-flow patterns. This synergy ensures an optimal balance between precision, efficiency, and the depth of exploration, addressing the critical challenges of feasibility of path coverage.

3.2 LLVM Loop Terminology

To explain the proposed approaches, it is essential to introduce LLVM loop terminology [17], as Rust is built upon the LLVM framework and handling the loops is the core idea of this work. LLVM defines a **Loop** as a subset of nodes from a control flow graph (CFG) with the following properties:

- prop1* The induced subgraph of the loop is **strongly connected**, meaning there exists a path between any two nodes within the loop.
- prop2* A loop has a single, unique **header**. The loop header is a node that dominates all other nodes in the loop and serves as the entry point for the loop. It is the only node in the loop that has predecessors outside the loop. A node can be the header of at most one loop, and loops can be identified or represented by their headers.
- prop3* A loop is the **maximum subset** of nodes in the CFG satisfying the above two properties.

In addition, loops in LLVM can be nested, meaning that the node set of one loop can be a subset of another loop's node set with a different loop header. Understanding the relationships between loops and their components is critical for explaining my DAGification and Path Reduction algorithms.

LLVM assigns special names to certain nodes and edges in the Control Flow Graph (CFG). For clarity and ease of explanation in this thesis, I introduce additional terms alongside LLVM's definitions. These terms are applied based on loops that have been simplified into canonical forms using the LoopSimplify algorithm [18].

Nodes in a loop are classified into three types based on their roles: **Latch**, **Header**, and **Normal** nodes. LLVM defines a **Latch Node** as a node that contains a back edge. Latch nodes manage the control flow required for a loop to iterate. A **Header Node**, as explained in *prop2*, is the unique node that dominates all other nodes in the loop and serves as the entry point. In this paper, any node within a loop that is neither a Latch nor a Header is referred to as a **Normal Node**.

Nodes can also be classified based on their relationships to loops. For instance, a node outside a loop that has an edge into the loop header is called an **Entering Block** (or **loop predecessor**). In the canonical loop form enforced by `LoopSimplify`, there must be only one entering block per loop, and its only edge is to the **header**. In this case, the entering block is referred to as the loop's **Preheader**. The preheader dominates the loop but is not part of the loop itself. On the other hand, a node inside the loop with an edge to a node outside the loop is referred to as an **Exiting Block**, while its target is called an **Exit Block**.

Edges are also classified based on their roles within the loop. LLVM defines a **Back Edge** as an edge where the source is inside the loop and the target is the loop header. In this paper, I introduce the term **Forward Edge** to describe the opposite: an edge from a node inside the loop to another node that does not return to the header.

Edges can also be named based on their roles between loops. LLVM refer to an edge from inside the loop to a node outside the loop as **Exiting Edge**. Conversely, an edge from a node outside the loop to the loop header is referred to in this paper as an **Entering Edge**.

The above terminology, both from LLVM and introduced in this thesis will be used for describing loops and their components in the context of the DAGification and Path Reduction algorithms.

Chapter 4

Approach

4.1 Overall Framework

Figure 4.1 illustrates the overall framework of the proposed Rust-based coverage-guided fuzzing system. This fuzzer integrates tightly with the Rust toolchain, leveraging MIRI to extract key execution information for path coverage analysis. The fuzzing engine is divided into several/two key components: the pre-processing phase and post-processing.

The system begins with a pre-processing phase, which includes static analysis, with DAG-ification performed during this phase. In this phase, the control flow graph (CFG) of the Program Under Test (PUT) is analyzed and prepared for fuzzing. Using MIRI, the CFG is collected, which provides structural information about the code. We call this process `static dump`. These outputs are stored as a single file and built into a unified program graph, which is a unified representation of the entire program, combining intra-procedural(CFG) and inter-procedural(call graph) flows. I will call this the *execution graph* in my thesis. The execution graph is then passed to the `DAGification` algorithm, one of the core contributions of this work. The name `DAGification` was chosen since in this process we attempt to transform the cyclic control flow graph into a directed acyclic graph (DAG). As a result of `DAGification`, loop information map (`LoopInfo`) is extracted, which marks each node as a *header* node, the *latch* node, or *normal* node. This result becomes critical for subsequent stages.

In the fuzzing loop, the system performs repetitive test case generation and PUT execution, just like general fuzzing structure. MIRI executes the test case to collect/dump a program execution trace. This is implemented by modifying the MIRI codebase. This

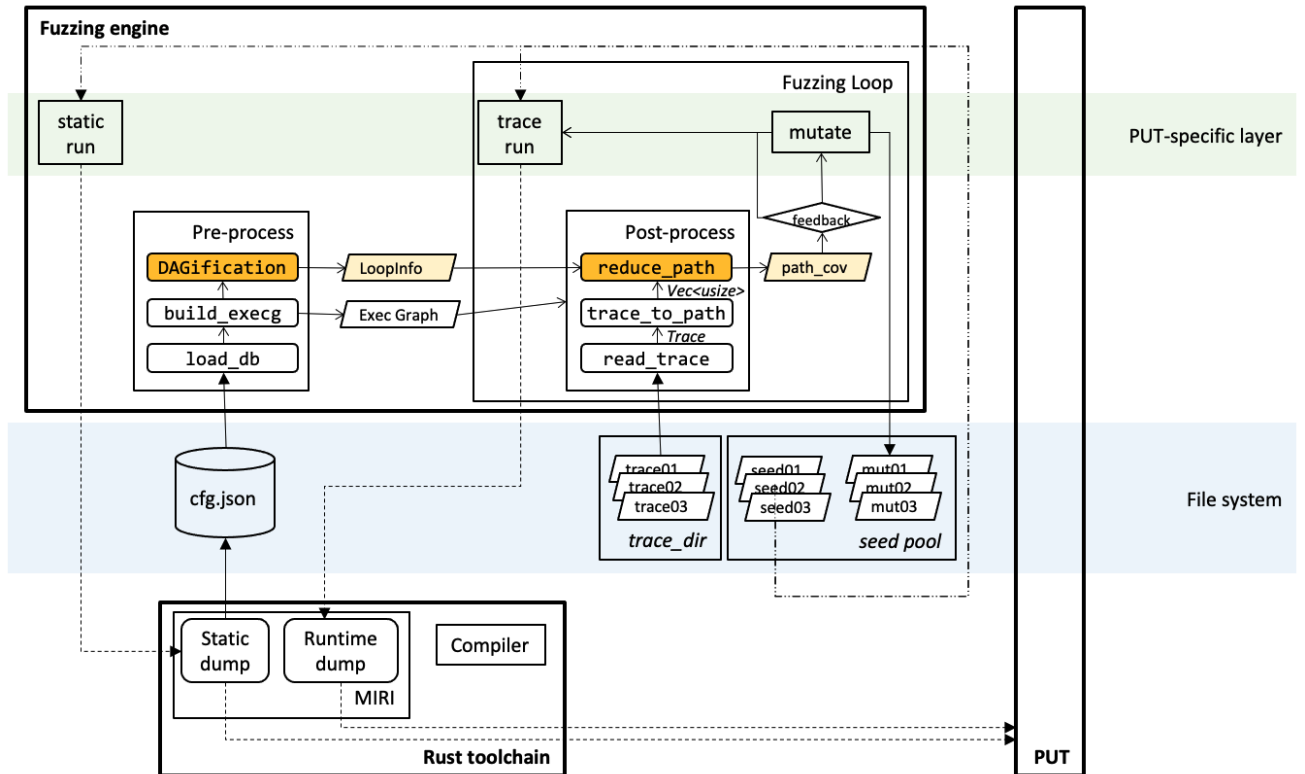


Figure 4.1: Overall Framework

trace is fed into the `trace_to_path` module, which maps the execution trace to a set of compact and representative paths based on the execution graph. The paths are then processed using the `path reduction` algorithm, the second key contribution of this work. The reduced paths is used as the path coverage, which provides feedback for the fuzzing process.

The file system is used to manage inputs and outputs. The seed pool stores initial and mutated inputs, while the `trace_dir` stores execution traces generated by MIRI. Mutated inputs and traces are continuously updated as the fuzzing loop progresses, ensuring efficient exploration of the PUT’s code.

By integrating DAGification and `path reduction` algorithms into a coverage-guided fuzzing loop, the system achieves efficient path exploration. In the following sections, we provide additional details and demonstrate how these core algorithms/approaches satisfy LLVM’s loop terminology. In the next chapter, 5.2.2, we explain how this was implemented

in Rust with minimal instrumentation overhead.

4.2 DAGification Algorithm

DAGification identifies the individual loops within the execution graph while also classifying the type of each node within these loops. This approach systematically analyze and transform the execution graph to simplify its structure by identifying loops, assigning unique identifiers to each loop, and splitting down the graph through the removal of back edges.

The process of DAGification transforms the cyclic CFG into a DAG by systematically identifying and simplifying loops. Loops can also be nested inside each other, making the decomposition of loops non-trivial. As we mention in background LLVM loop terminology section, a loop in LLVM is defined as an induced subgraph that:

prop1 It is **strongly connected**.

prop2 It has a **unique header** that dominates all nodes in the loop.

prop3 It represents the **maximum subset** of nodes with *prop1* and *prop2*.

The DAGification process ensures that these properties remain intact by employing a recursive approach that leverages three steps below.

step1 Identify (all) SCCs using kosaraju algorithm

step2 Simplify Loops using LLVM LoopSimplify

step3 Decompose Loops: remove the back edge

the DAGification process recursively processes the remaining loops using the same three-step approach until no further cyclic structure is found, ensuring that all loops are correctly identified and decomposed. This recursive approach guarantees that the graph is fully transformed into a DAG without violating the fundamental properties of LLVM loops.

In the next sections, I explain how each steps work and demonstrate how it can remain the Loop's properties intact. *prop1* and *prop3* of Loop can be satisfied by using exist-ing SCC algorithm, kosaraju algorithm. I show *prop2* is satisfied by adapting LLVM's LoopSimplify algorithm.

4.2.1 Kosaraju’s Algorithm

The first step in the DAGification process involves identifying all Strongly Connected Components (SCCs) in the CFG [19]. An SCC is a maximal subgraph in which every node is reachable from every other node. Kosaraju’s algorithm is employed for this task due to its efficiency and simplicity. This algorithm operates in two passes: First, it performs a depth-first search (DFS) on the original graph and record the finishing order of nodes. Second, it reverses the edges of the graph and perform a second DFS based on the finishing order obtained in the first pass. Each DFS tree corresponds to an SCC.

This step identifies all SCCs, which are potential loops in the CFG, and prepares them for further processing. Kosaraju’s algorithm ensures that all strongly connected subgraphs are accurately identified without altering the underlying structure of the graph.

4.2.2 LLVM LoopSimplify

Once the SCCs are identified, each Loop is simplified using the LLVM LoopSimplify transformation. The goal of this step is to normalize the loops by converting potentially complex structures into a simplified loop with a single latch and preheaders [20]. Specifically, LoopSimplify ensures the following:

1. Each loop has a dedicated preheader, making loop entry explicit.
2. Loops with multiple back edges are converted into loops with a single back edge, simplifying the loop’s structure.

This normalization process is crucial for enabling subsequent transformations, where the single back edge is removed.

4.2.3 Decomposing Loops

The final step in the DAGification process involves decomposing the loops by temporarily removing their back edges. A back edge is defined as an edge that connects a node inside the loop to the loop header, enabling cyclic behavior. Removing these back edges eliminates cycles from the graph, transforming it into a Directed Acyclic Graph (DAG). Since LoopSimplify in step 2 guarantees that the loop is in a canonical form, in step 3 we can simply remove one edge without affecting the structure.

To summarize, In *step 2*, by using existing SCC algorithm, *prop 1* and *prop 3* is demonstrated. *step 2* leverages LLVM’s existing `LoopSimplify` pass so that the *step 3* can avoid introducing inconsistencies or deviations from LLVM’s loop definitions’s *prop 2*.

4.3 Path Reduction Algorithm

Path reduction is a deterministic algorithm designed to adjust the length of execution paths by limiting the number of iterations recorded within loops, based on a user-defined unroll parameter. The core idea is to maintain a balance between the granularity and efficiency, avoiding overly long paths caused by excessive iterations within loops. The user specifies the maximum number of iterations to be recorded for each loop. If the number of back edges in a loop exceeds this limit, subsequent paths within the loop are discarded to prevent redundant information.

As introduced in the LLVM loop terminology section 3.2, LLVM categorizes edges in the control flow graph based on their roles within a loop and between loops. Edges within a loop are classified as either **Back Edge** or **Forward Edge**. Additionally, nested loops introduce edges classified as **Entering Edge** and **Exiting Edge**. *Path reduction* determines the role of a given edge both within a loop and between loops based on this terminologies to adjust the number of iterations.

The algorithm categorizes the edge into one of three categories by comparing the sizes of two stacks: **Entering Edge**, **Exiting Edge**, or **Within-the-same-loop Edge**. These stacks represent the nesting levels of the nodes within the loop hierarchy, where (**s**) is a stack associated with the source node and (**t**) is a stack associated with the target node of a given edge. Based on the difference $k = \text{sidx} - \text{tidx}$, where $\text{sidx} = \mathbf{s}.\text{len}() - 1$ and $\text{tidx} = \mathbf{t}.\text{len}() - 1$, the operations for handling loop is decided. Only if the source and target nodes are in the same loop, the algorithm examines the content of the stacks to determine whether the edge is a **Back Edge** or a **Forward Edge**.

Below, we explain how these three cases can be further divided into subcategories and why other scenarios are theoretically impossible

4.3.1 Case 1 (Entering Edge): $k < 0$

The scenario of $k < -1$ is theoretically impossible. This would imply a transition directly from an outer loop to a deeply nested inner loop, bypassing intermediate loops. For

instance, suppose `LoopC` is nested within `LoopB`, and `LoopB` is nested within `LoopA`. A transition directly from `LoopA` to `LoopC` contradicts the properties enforced by DAGification. In the DAGification process, after eliminating the cyclic dependency in the outer loop, `LoopA`, the header of the next nested level, `LoopB`, can be identified. Similarly, after removing the cyclic dependency in `LoopB`, the header of `LoopC` can be identified. This ensures that the header of `LoopC` is always dominated by the header of `LoopB`. Consequently, an edge directly connecting a block in `LoopA` to the header of `LoopC` without passing through the header of `LoopB` cannot exist. Such an edge would violate the hierarchical dominance relationships established by `LoopSimplify` and the DAGification algorithm.

Therefore, Case 1 is redefined as $k = -1$. Here, the stack s is shorter by one element compared to the stack t . This indicates that the given edge is an **Entering Edge**, representing a transition into the next nested loop level.

4.3.2 Case 2: $k = 0$

When the stacks s and t have equal lengths ($k = 0$), two subcategories arise: either a transition within the same loop or a transition between two loops at the same nesting level.

Transition Between Loops:

If the two nodes are at the same loop level but not in the same loop, the edge represents an **Exiting Edge** followed by an **Entering Edge** into another loop. This transition occurs when exiting one loop and immediately entering another nested loop. The algorithm handles this case by performing the operations for Case 3 (Exiting edge) and Case 1 (Enter edge) sequentially.

Transition Within the Same Loop:

If the two nodes are in the same loop, the edge can be either a **Back Edge** (indicating an iteration) or a **Forward Edge** (indicating progression). The type is determined by examining the content of the stacks. If the edge is a forward edge, the algorithm records the basic block. If the edge is a back edge, the algorithm performs an operation to decide whether to record or discard the sequence of blocks traversed so far.

4.3.3 Case 3 (Exiting Edge): $k > 0$

Unlike $k < -1$ in Case 1, where such scenarios are impossible, $k > 1$ is valid. This reflects the possibility of exiting multiple nested loop levels simultaneously. For example, an edge may represent an **Exiting Edge** from a block in LoopC directly transitioning to an **Exit Block** in LoopA.

The algorithm handles this by performing the specific operation for exiting edge and decrementing $sidx$ by 1 repeatedly until $k = 0$. Once $k = 0$, the edge is reclassified as Case 2, and the corresponding operations are applied.

The deterministic nature of this algorithm ensures that all paths are reduced consistently without violating LLVM loop properties. Due to the simplifications introduced by LLVM's `LoopSimplify`, each loop is normalized into a structure with a single latch and preheader, ensuring well-defined transitions between loops. This guarantees that the algorithm operates within the valid categories of edges. Also, the comparison of two stacks is computationally efficient, involving straightforward operations such as stack length comparisons and updates.

In the next chapter (5.2.2), we provide a detailed explanation of how each case is handled with the specific operations to reduce the actual path effectively.

Chapter 5

Implementation

This chapter describes the implementation details of the algorithms introduced in the Approach chapter (4.3.3), specifically focusing on their application in Rust programming language. The implementation is divided into two sections: Static Analysis and Runtime Analysis. The Static Analysis section discusses the pre-processing phase, which includes the implementation of static CFG dump and DAGification, while the Runtime Analysis section explains the post-processing phase, which covers runtime trace dump and Path Reduction.

In this work, as illustrated in Figure 4.1, we introduced two features to the Rust Compiler and MIRI tool: the `static CFG dump` and the `runtime trace dump`. The DAGification and Path Reduction algorithms are implemented in `fuzzing_engine` component. The DAGification takes the execution graph as input and outputs a loop information table, while the path reduction algorithm takes the loop information table as input and generates a bounded path. These components are integral to achieving efficient path coverage-based fuzzing by reducing redundant loops and enhancing the effectiveness of fuzzing.

5.1 Static Analysis

In the context of fuzzing, the static analysis phase is a pre-processing step performed only once per target program. As such, it has minimal impact on the overall fuzzing performance.

5.1.1 Static CFG Dump

To perform static analysis, we modified the Rust compiler toolchain to dump the CFGs and call graph of the target program into the file system. These graphs are then used to build an execution graph, serving as the foundation for further analysis.

The implementation required modifications to the Rust compiler codebase, specifically in the `compiler/rustc_middle/src/ty` module. Approximately 1100 lines of code were added to enable this functionality. The static CFG dump captures both intra-procedural control flow (CFGs) and inter-procedural function call relationships (call graph), which are then combined into a unified execution graph.

5.1.2 DAGification

The DAGification process transforms the cyclic execution graph into a Directed Acyclic Graph (DAG) and outputs a table called the *Loop Information Map* to provide a hierarchical representation of loops. This map is a key-value structure where the keys represent node indices in the execution graph, and the values are stacks containing detailed loop information.

Algorithm 1 DAG-ification

Require: Input: Execution Graph ($G = (V, E)$),

Require: Output: DAG information map (*loop_info_map*)

procedure DAGIFICATION(G, scc_list)

loop_info_map = new()

scc_id = 0

scc_list \leftarrow KOSARAJU_SCC(G)

 DAGIFICATION_RECURSIVE(*scc_list*, *scc_id*, G , *loop_info_map*)

return *loop_info_map*

end procedure

Figure 5.1: DAGification

Figure 5.1 and Figure 5.2 provide the pseudocode for the DAGification process, which consists of two interdependent algorithms: **DAGification** and **DAGification-Recursive**. Together, these algorithms detect and resolve cycles in the graph, ultimately producing an acyclic representation for further runtime analysis.

Algorithm 2 DAG-ification-recursive

Require: Intermediate Output: modified Graph (G')

```
procedure DAGIFICATION(scc_list, scc_id, G, loop_info_map)  
  for all scc  $\in$  scc_list do  
    loop_header  $\leftarrow$  GET_HEADER(scc)  
    cycle  $\leftarrow$  HAS_CYCLE(scc)  
    if cycle then  
      Stage One: Mark each node type  
      loop_info_map.insert(loop_header, LoopInfo(scc_id, Header))  
      latch  $\leftarrow$  GET_SINGLE_LATCH(loop_header)  
      loop_info_map.insert(latch, LoopInfo(scc_id, Latch))  
      for all node  $\in$  scc do  
        if node  $\neq$  latch and node  $\neq$  loop_header then  
          loop_info_map.insert(node, LoopInfo(scc_id, Normal))  
        end if  
      end for  
      Stage Two: Break down into DAG ( $G'$ )  
       $E' \leftarrow E - \text{EDGE}(\text{latch}, \text{loop\_header})$  ▷ Remove back-edge  
       $G' = (V, E')$   
      Stage Three: DAG-ify Recursively  
      scc_id $+$  = 1  
      sub_scc_list  $\leftarrow$  KOSARAJU_SCC( $G'$ )  
      DAGIFICATION_RECURSIVE(sub_scc_list, scc_id,  $G'$ , loop_info_map)  
    end if  
  end for  
end procedure
```

Figure 5.2: DAGification-Recursive

DAGification Algorithm The **DAGification** algorithm takes a graph $G = (V, E)$, where V is a set of nodes and E is a set of edges. It begins by identifying strongly connected components (SCCs) in the graph using Kosaraju’s algorithm. SCCs represent subgraphs that contain cycles and are the primary targets for transformation. The algorithm initializes `loop_info_map` and identifier (`scc_id`). It calls the **DAGification-Recursive** function to populate the `loop_info_map` with specific loop details.

DAGification-Recursive Algorithm As described in chapter 4.3.3, the DAGification-Recursive algorithm is composed of three key steps: identifying SCCs, simplifying loops, and decomposing loops with node marking.

In Figure 5.2, Stage One and Stage Two correspond to marking nodes and decomposing loops, respectively. To decompose a loop, the algorithm removes back edges, resulting in

an intermediate subgraph G' . This subgraph is a simplified, acyclic representation of the original SCC. The step labeled `KOSARAJU_SCC` corresponds to the identification of SCCs, while `GET_SINGLE_LATCH` represents the loop simplification process.

The recursive nature of the algorithm ensures that nested loops and complex hierarchies are systematically handled, allowing the graph to be fully transformed into DAG.

5.2 Runtime Analysis

The runtime analysis phase is a post-processing step performed within the fuzzing loop, making significantly impacts fuzzing performance. To minimize this impact, we made efforts to introduce only minimal instrumentation to the MIRI codebase for dumping execution traces. The Path Reduction algorithm is implemented in the `fuzzing_engine` component and is designed to process each edge efficiently by comparing two stacks associated with the source and target nodes. This lightweight design ensures that classifying edges and reducing paths do not cause significant performance degradation.

5.2.1 Runtime Trace Dump

We modified MIRI to dump the execution path during program execution. It was implemented by adding approximately 40 lines of code to `src/tools/miri/src`. Additionally, we added 164 lines to `compiler/rustc_const_eval/src/interpret`, introducing functions such as `push_bb`, `push_call`, `merge_trace`, and `runtime_dump`.

5.2.2 Path Reduction

The path reduction algorithm processes the dumped execution trace to generate a bounded path, eliminating redundancy caused by excessive loop iterations. This ensures that the fuzzer can focus on exploring a wide variety of paths without being overly concentrated on specific loops.

Five Pre-defined Operations The algorithm employs five predefined operations: `Push`, `Pop` and `Merge`, `Add`, `Extend`, and `Extend or Discard`. These operations are implemented using two helper structures: `PathBlock`, which represents individual path segments, and `PathStack`, which manages the hierarchy of path segments. `PathBlock` con-

tains `prefix` which is the sequences of basic block id and `count_map` which maps `prefix` to it's appearance count.

Below, I detail the five predefined operations:

- **Push:** This operation adds a new and empty `PathBlock` to the `PathStack`. Each `PathBlock` corresponds to a distinct Loop(SCC). When encountering an entering edge, the algorithm pushes a new `PathBlock` to represent the transition into a new loop. This ensures that loop hierarchies are preserved.
- **Pop and Merge:** When exiting a loop, the algorithm pops the top `PathBlock` from the `PathStack` and merges its contents with the next `PathBlock`. The prefix from the exiting `PathBlock` is appended to the prefix of the next `PathBlock` in the stack, and the stack index is decremented.
- **Add:** This operation appends the target basic block's ID to the prefix of the current top `PathBlock` on the `PathStack`. It is used to extend the path within the current loop level. This operation is invoked whenever the algorithm encounters a forward edge or a transition within the same loop level.
- **Extend:** This operation moves the current prefix of the top `PathBlock` to the main path and clears the prefix. It consolidates the current execution context and prepares the `PathBlock` for the subsequent edge.
- **Extend or Discard:** This operation is key operation to reduce the path. To enforce the user-defined unrolling parameter, this operation checks the count of the current prefix in the `count_map`. If the count is below the limit, the algorithm performs an **Extend** operation. Otherwise, the prefix is discarded, ensuring that redundant iterations are eliminated from the reduced path. The count is updated accordingly.

Application of Operations As shown in Figure 5.3, the path reduction algorithm invokes these operations individually or in combination based on the edge type.

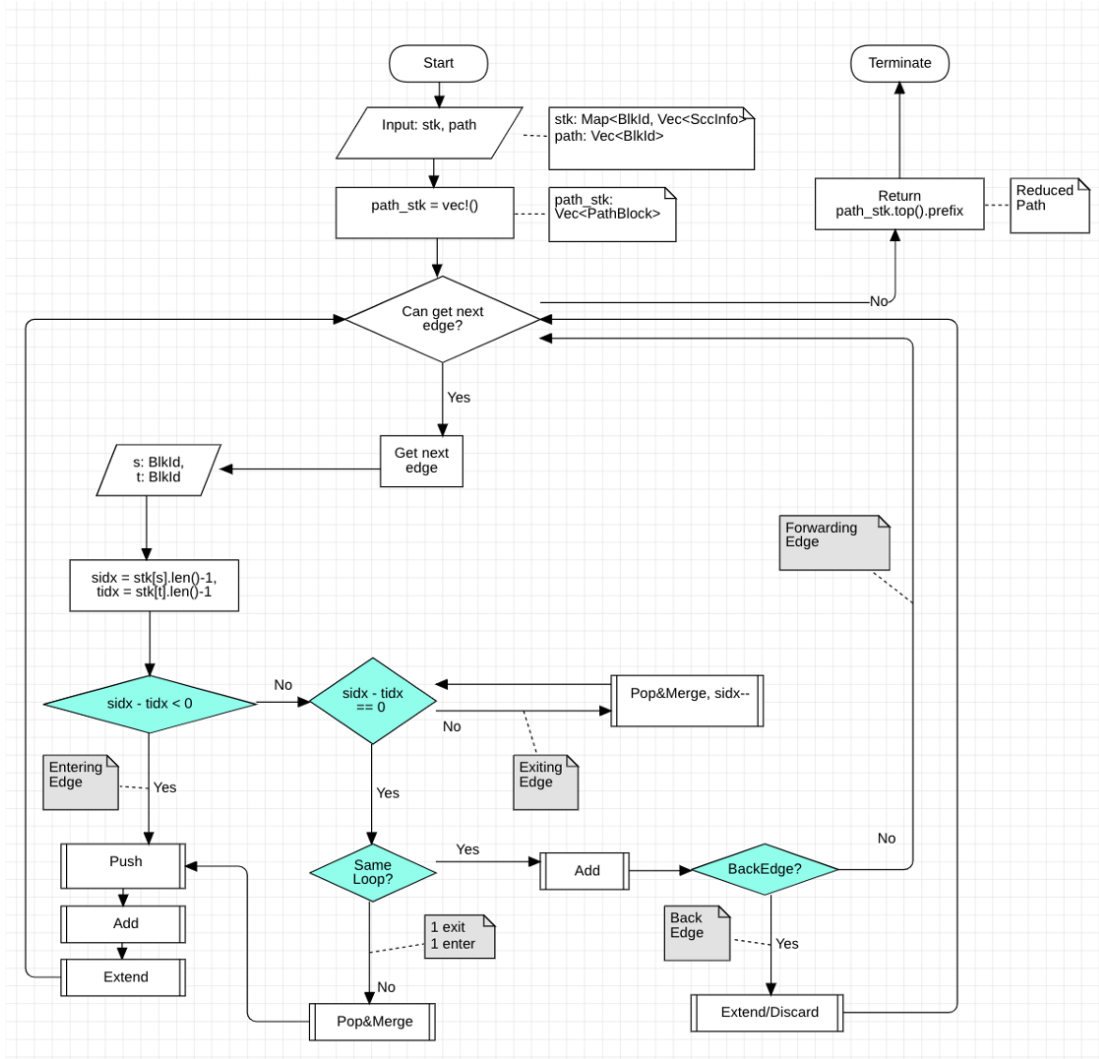


Figure 5.3: Path Reduction Flowchart

The algorithm begins by classifying each edge in the execution trace into one of the three cases introduced in chapter 4.3.3: **Entering**, **Exiting**, and **Within-the-same-loop**. Depending on the edge type, the appropriate operations are applied to record or discard the subset of path. If no additional edges are available, the algorithm terminates and

returns a reduced path, which is stored in the `PathStack` structure. Otherwise, it retrieves the next edge in the path for further examination.

Case 1: $k < 0$ The edge is classified as an **Entering Edge**. This triggers a sequence of **Push**, **Add** and **Extend** operations.

Case 2: $k = 0$ The algorithm checks whether both nodes are within the same loop by comparing their `SCCID`.

If they belong to the same loop, the algorithm determines whether the edge is a **Back Edge** or a **Forward Edge** by examining the content of the `PathBlock`. For **Back Edges**, the **Extend** or **Discard** operation is invoked to enforce the iteration limit. For **Forward Edges**, the **Add** operation is performed, and the algorithm continues examining subsequent edges.

If the source and target nodes are not in the same loop, the edge represents a transition involving both an exit from one loop and an entry into another. In this case, **Pop and Merge** is processed for exiting operation and then the same sequence of operations as in Case 1 (**Push**, **Add**, and **Extend**) is followed for the entering operation.

Case 3: $k > 0$ The edge is classified as an **Exiting Edge**. This scenario can involve exiting multiple nested levels consecutively. The algorithm performs the **Pop and Merge** operation repeatedly, decrementing the level of source node stack until the source and target nodes are within the same loop segment level. Once the nodes are at the same level, the algorithm transitions to handling the edge using the operations from Case 2.

This structured approach ensures that path reduction is handled systematically for all edge types, maintaining consistency and adhering to the constraints introduced by LLVM's loop properties.

Chapter 6

Evaluation

To measure the effectiveness of the proposed fuzzing technique, this chapter presents an evaluation using my testing harness.

The experiments aim to address the following research questions:

- **RQ1.** Bug-finding ability in real-world Rust projects.
- **RQ2.** Validity of the loop unrolling parameter.
- **RQ3.** Coverage growth comparison.

6.1 Evaluation Setup

6.1.1 Dataset

The dataset used in this evaluation consists of four Rust libraries selected from `crates.io`, the official Rust package registry, along with the Move bytecode-verifier component from the Aptos Move compiler.

Table 6.1 presents the statistics of the dataset, summarizing the size and complexity of the target libraries. This dataset provides a realistic testing ground for evaluating the prototype fuzzer’s performance. **CFG** refers to the number of functions that form the execution graph. The **Block** and **Edge** columns represent the total number of blocks and edges in these CFGs. **Loop** refers to the total number of loops in the CFGs, while **Loop***

PUT	CFG	Block	Edge	Loop	Loop*	Seeds
time	64	1037	1111	0	0	201
chrono	200	1999	2449	2	2	160
fixedbitset	145	1039	1296	26	41	200
url	351	5204	6941	84	2518	143
Move	2353	28013	38398	378	2960	220

Table 6.1: Summary of the dataset

accounts for loops in the execution graph. For example, if one CFG contains 3 loops and this function is called 4 times from different call sites, Loop is 3, but Loop* is 12. The Seeds column indicates the number of seed inputs.

6.1.2 Test Oracle

In Rust, the primary error-handling mechanism is the `panic!` macro. Consequently, the dataset was selected to include code paths with potential panics, enabling the test harness to detect issues in error-prone sections of code. The occurrence of a panic serves as the test oracle, marking failure points or bugs during fuzzing.

6.1.3 Experiment Infrastructure

Two experimental setups were used: `thread150` and `thread3`.

- `thread150`: Conducted on an Ubuntu LTS system with an 80 core Intel CPU, and 1 TB of RAM. This setup used 150 worker threads and 4 producer threads to generate new inputs.
- `thread3`: Conducted on an Ubuntu LTS system with an 8 core Intel CPU, and 16 GB of RAM. This setup used 3 worker threads and 1 producer thread.

6.2 Bug-Finding Ability (RQ1)

In RQ1, the primary goal is to demonstrate the effectiveness of the proposed fuzzing technique in discovering bugs located in deep execution paths of real-world Rust projects with no previously known vulnerabilities. Two case studies are introduced and analyzed.

6.2.1 Case Study 1: Move Bytecode Verifier

Motivation The specific challenges associated with the `Move` target motivated the application of our path coverage-based fuzzing techniques. Since the target function `verify_module` contains a series of sequential tests, a wide variety of paths can emerge depending on which stage of the function fails. Path coverage may increase the likelihood of discovering bugs compared to block or edge coverage.

Goal This case study aims to demonstrate that the algorithm can be applied to programs with extensive loops and large codebase and, if possible, identify actual bugs.

Result As shown in Table 6.1, `Move` PUT is of considerable size. Its execution graph consists of 2353 CFGs, forming 2960 loops in total. The PUT was executed using the `thread150` setup for 12 days (approximately 291 hours). During the experiment, 2,920,491 executions occurred, and 352,216 new paths were explored, but no crashes were found. This highlights the limitations of the current prototype testing harness.

Limitations To achieve more useful results, two improvements are proposed:

- **Mutation Strategy:** The `verify_module` function strictly examines a custom structure, `CompiledModule`. In this process, inputs that fail early in the verification process result in short paths, limiting the exploration of subsequent functions. A more sophisticated mutator could avoid generating inputs that fail early.
- **Seed Selection Strategy** [21]: Prioritizing inputs with longer paths for mutation could improve the efficiency of fuzzing.

6.2.2 Case Study 2: Real-World Bug Discovery

The fuzzer successfully uncovered one unknown bug in the `url` [22] crate and identified a discrepancy in the bench tests of the `chrono` [23] crate.

Result In the `url` crate, an assertion failure was discovered during the slicing process in the `index` function located in `url/src/slicing.rs`. This issue was reported on GitHub and is currently awaiting a response.

In the `chrono` crate, a crash was detected during string formatting in a fuzz target based on the crate’s bench tests. This was also reported on GitHub, but it turned out to be a discrepancy within the bench tests rather than a bug in the function itself.

These results demonstrate the capability of the proposed fuzzing technique to detect new, real-world issues in Rust projects within a reasonable timeframe, showcasing its practical utility.

6.3 Validity of Loop Unrolling Parameter (RQ2)

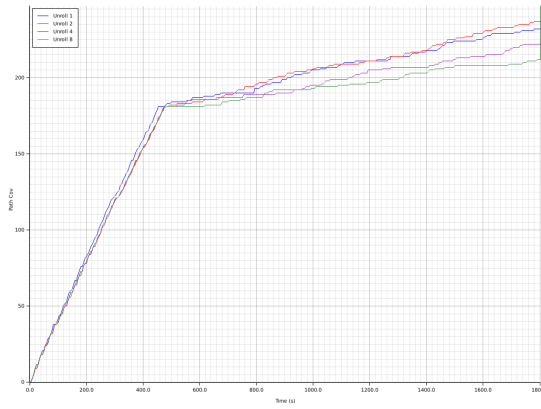
In this section the impact of varying the loop unrolling parameter on bounded path coverage growth was measured and visualized. The analysis focuses on comparing the impact of pre-defined unrolling parameters on PUT with and without loops in the execution graph.

PUT	Loop	1	2	4	8
<code>time</code>	0	566	545	583	525
<code>fixedbitset</code>	41	75	199	518	772

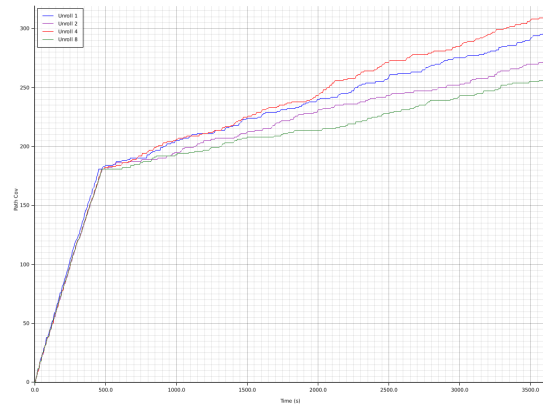
Table 6.2: Bounded path coverage size for each parameter

Dataset From Table 6.1, we selected `time` to represent a PUT without a loop, and `fixedbitset` to represent a PUT with a loop. As shown in Table 6.2, Each PUT was tested with unrolling parameters of 1, 2, 4, and 8. The coverage size in the table is recorded at 12600 seconds (approximately 3.5 hour) elapsed.

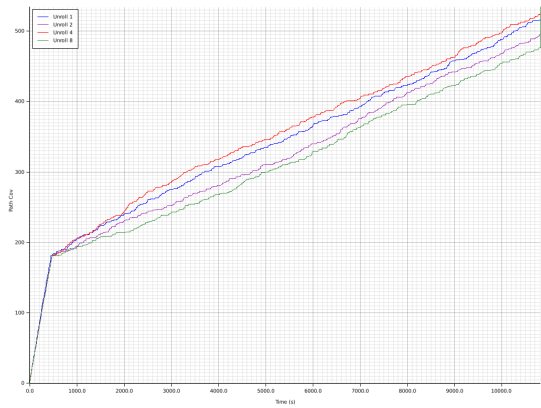
Result The CFG of `time` target does not contain any loop, and consequently, `Loop` and `Loop*` in Table 6.1 are also zero. Theoretically, if there are no loops, there should be no difference among the four parameters. In Figure 6.1, the x-axis represents time, each colored graph corresponds to the path coverage growth for the respective unroll parameters (1, 2, 4, and 8). The path coverage growth graphs exhibit very little difference from one another. As a demonstration of this, in Figure 6.1a, the graphs even overlap and intertwine inconsistently, highlighting the lack of significant impact of the unroll parameter in the absence of loops.



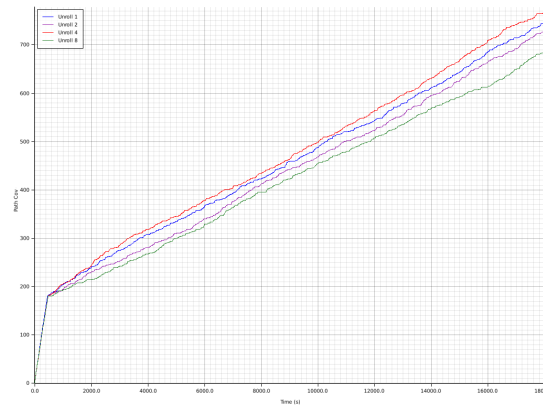
(a) 0.5 hour



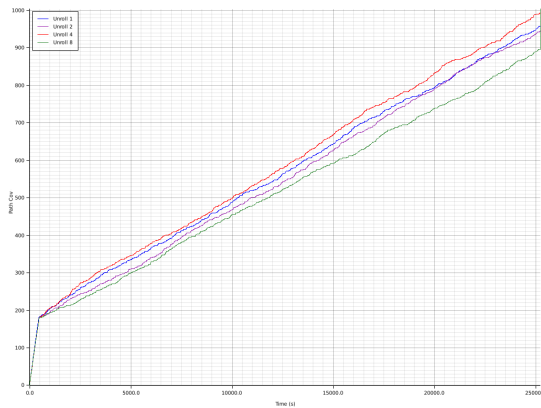
(b) 1 hour



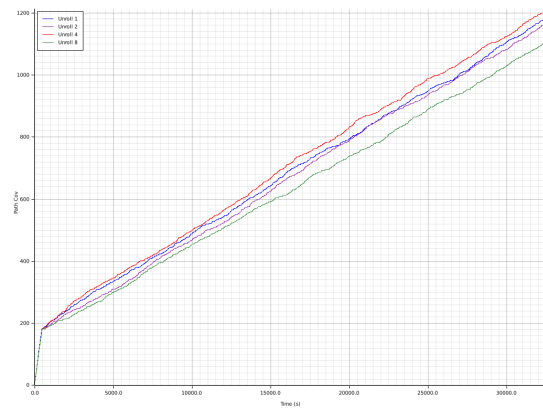
(c) 3 hour



(d) 5 hours

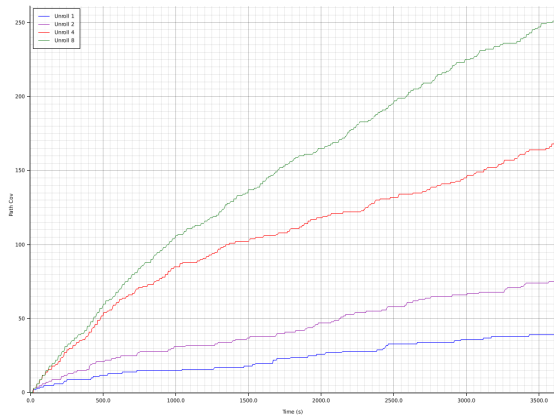


(e) 7 hour

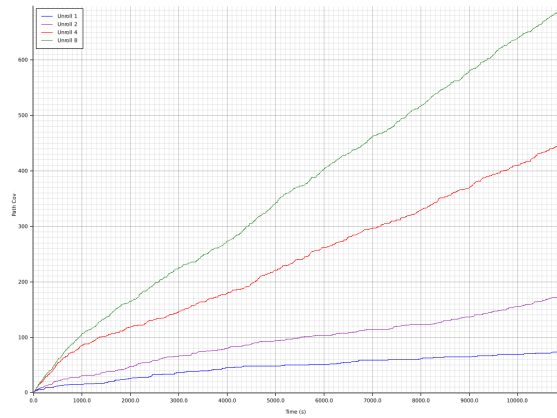


(f) 9 hours

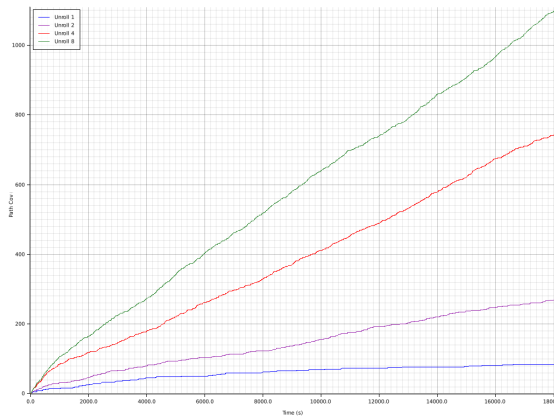
Figure 6.1: RQ2 time



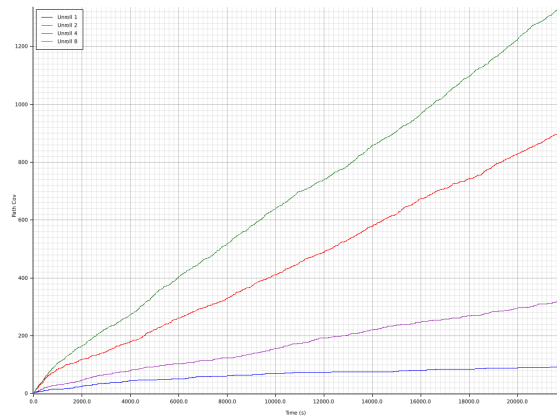
(a) 1 hour



(b) 3 hours



(c) 5 hours



(d) 6 hours

Figure 6.2: RQ2 fixedbitset

Compared to Figure 6.1, Figure 6.2 shows a noticeable difference for each parameters. The growth trends in Figure 6.2 suggest that as the loop unrolling parameter increases, path coverage growth also increases since an increase in the unrolling parameter allows the Path Reduction algorithm greater flexibility. Consequently, the number of possible unique paths increases. Thus, within a range where this flexibility does not hinder the exploration of new paths, a larger parameter leads to a more significant increase in path coverage size.

Limitation In the results for `time` library, slight differences between the lines are still observed. This may be attributed to the inherent randomness of fuzz testing, as the results

can vary slightly based on mutated inputs. As such, there is always a potential error range. To improve the reliability of the findings, the same experiment could be repeated multiple times, and the average results could be calculated.

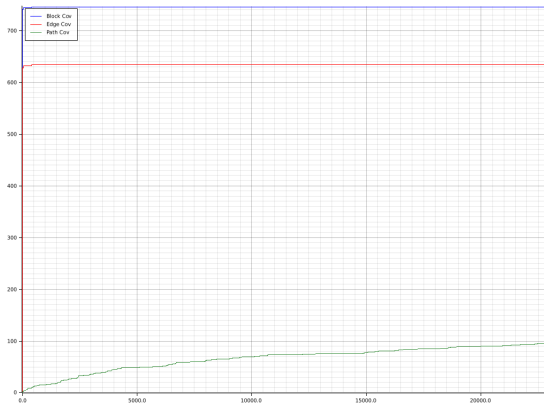
In addition, the two PUT used in this evaluation were one with no loops and one with a small number of loops. For more reliable experimental results, we plan future work to evaluate PUTs with a very large loop count, such as `url` or `move`.

6.4 Coverage Growth Comparison (RQ3)

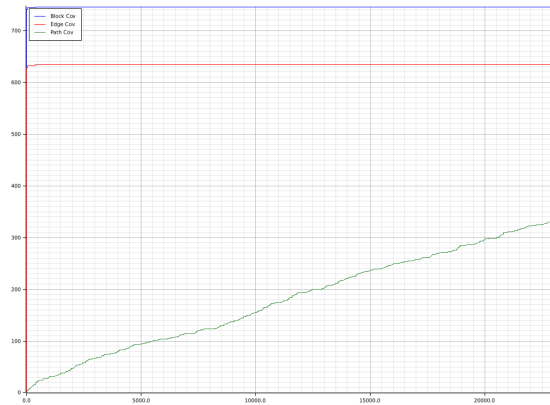
To investigate whether bounded path coverage can provide a more thorough exploration of the API surface, we compared the growth of block, edge, and bounded path coverage over time.

Goal The primary goal is to analyze the growth trends of bounded path coverage, block coverage, and edge coverage. Additionally, we aim to demonstrate that bounded path coverage can reach saturation, thereby preventing the path explosion issue often associated with full path coverage.

Result We present the result of `fixedbitset` evaluated with `thread150` setup with an unrolling parameter of 2. Figure 6.3 reveals an interesting finding: Bounded path coverage of `fixedbitset` reaches saturation with fewer paths compared to block and edge coverage when the unrolling parameter is set to 1. Similarly, another interesting fact is that, unlike `fixedbitset` PUT, in other PUTs such as `url` and `time`, the edge coverage is usually larger than block coverage size. Additionally, with the unrolling parameter 2, the possibility of saturation is observed. Figures 6.3a and 6.3b demonstrate that the growth trend for path coverage eventually saturates with unroll parameter 1 and 2. However, to clearly justify that proposed metrics is feasible, the coverage metrics should be saturated with larger unrolling parameter.



(a) unroll=1



(b) unroll=2



(c) unroll=4



(d) unroll=8

Figure 6.3: RQ3 fixedbitset

Limitation We need to provide stronger evidence of saturation to demonstrate that bounded path coverage can prevent the path explosion problem, by evaluating on larger PUTs. On smaller PUT such as `time` and `fixedbitset`, we can observe the saturation with unrolling parameter set to 2. However, this result is not enough to conclude that our metric can prevent path explosion. Future work could explore extended fuzzing period or alternative unrolling parameters to observe whether saturation occurs under different conditions.

Additionally, to provide a more comprehensive perspective, another experiment could be conducted to evaluate the effectiveness of path coverage as a feedback metric. This

experiment would compare path coverage against block and edge coverage as feedback mechanisms. Using the same dataset as RQ3, the feedback mechanism of the fuzzer would be changed from path coverage to block and edge coverage. The same metrics—changes in block, edge, and path coverage—would be recorded to assess the relative effectiveness of these feedback strategies.

Chapter 7

Future Work

This chapter summarizes the limitations identified during the evaluation and proposes directions for future work to address these issues and extend the capabilities of the proposed fuzzing framework.

Stable Evaluation Results through Averaging To mitigate the impact of randomness on the experimental outcomes, it is recommended to conduct each set of experiments multiple times and use the average of the results as the final metric [24]. This approach would provide more stable and reliable evaluation results, reducing variability caused by random inputs.

Developing a Generalized Test Harness for Benchmark Comparison The fuzzing engine used in the current evaluation is a prototype, which limited the inclusion of multiple PUTs. A more generalized test harness is currently under development. Once completed, this will enable evaluations across a diverse range of benchmarks, yielding more reliable and comprehensive results and facilitating comparisons with other fuzzing tools.

Improving Test Harness Efficiency One of the primary limitations observed during the evaluation was the test harness's inefficiency in fully leveraging the fuzzing framework's potential. Specifically, the mutation and seed selection strategies require refinement. Enhancing these aspects would allow the framework to generate more effective inputs and explore deeper paths.

Extending the Test Oracle Expanding the capabilities of the test oracle could further enhance the fuzzing framework. For instance, integrating memory leak detection and handling unsafe pointer cases could significantly improve the framework’s ability to detect complex vulnerabilities.

Verifying Saturation of Bounded Path Coverage on Large Targets One of the goals of this research is to demonstrate that bounded path coverage resolves the path explosion problem by achieving saturation. While this behavior was observed for small targets, such as `fixedbitset`, the results for larger targets remain inconclusive.

Evaluating Path Coverage as a Feedback Mechanism To establish the effectiveness of path coverage as a feedback mechanism, an experiment comparing the fuzzer’s performance when using path, block, and edge coverage as feedback metrics can be a future evaluation plan.

The proposed fuzzing framework demonstrates promising results in exploring execution paths and discovering bugs in Rust projects. However, the limitations identified during the evaluation point to several opportunities for future improvement. Addressing these challenges could significantly enhance the framework’s effectiveness and applicability, enabling it to better handle large and complex software systems.

Chapter 8

Conclusion

In this thesis, we propose a Rust-based path coverage-guided fuzzing, specifically designed to rigorously test targets requiring a more granular coverage metric. This approach was motivated by the growing adoption of Rust in security-critical domains, where projects demand thorough and meticulous testing.

State-of-the-art coverage metrics like edge coverage often fail to explore deeply nested or rare paths, leaving critical vulnerabilities undiscovered. While path coverage offers a more comprehensive approach, it suffers from path explosion, making exhaustive exploration impractical due to resource inefficiency. To address these challenges, this thesis introduced Bounded Path coverage, a novel metric that uses a configurable loop unrolling parameter to balance exploration depth with feasibility. This metric is achieved through the `DAGification` and `Path Reduction` algorithms.

Our prototype fuzzer was implemented and evaluated on real-world Rust projects to demonstrate its feasibility and effectiveness. The experimental evaluation addressed three research questions, focusing on bug-finding ability, the validity of the loop unrolling parameter, and performance comparison to block and edge coverage. The fuzzer uncovered critical issues, including an unknown bug in the `url` crate and a discrepancy in a benchmark test of the `chrono` crate.

In conclusion, the results of this work demonstrate that a Rust-specific, path-coverage guided fuzzer is both feasible and effective for testing real-world projects. Future work could focus on refining the test harness to improve fuzzing performance and facilitate easier comparisons with other coverage metrics.

References

- [1] Valentin J. M. Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey, 2020. Preprint available at <https://arxiv.org/abs/2001.03357>.
- [2] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS 2017)*. The Internet Society, 2017.
- [3] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP 2018)*, pages 679–696, San Francisco, CA, USA, 2018. IEEE.
- [4] Scott Olson. Miri: An interpreter for rust’s mid-level intermediate representation. Technical Report, 2016. Online; <https://github.com/rust-lang/miri>.
- [5] Miri Contributors. Miri: An interpreter for rust’s mid-level intermediate representation. <https://github.com/rust-lang/miri>, 2021. Accessed: 15 November 2024.
- [6] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018.
- [7] M. Zalewski. American fuzzy lop. <http://lcamtuf.coredump.cx/afl/>. Accessed: accessed 15 November 2024.
- [8] LLVM Project. libfuzzer – a library for coverage-guided fuzz testing. <https://llvm.org/docs/LibFuzzer.html>. Online; accessed 15 November 2024.
- [9] Google. Honggfuzz. <https://github.com/google/honggfuzz>, 2020. Online, accessed 15 November 2024.

- [10] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. Pata: Fuzzing with path aware taint analysis. In *Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP)*, pages 978–991. IEEE, 2022.
- [11] Shengbo Yan, Chenlu Wu, Hang Li, Wei Shao, and Chunfu Jia. Pathaff: Path-coverage assisted fuzzing. In *Proceedings of the 2020 ACM Conference on Computer and Communications Security (CCS '20)*. ACM, 2020.
- [12] Rust Fuzzing Authority. afl.rs: Fuzzing rust code with american fuzzy lop. <https://github.com/rust-fuzz/afl.rs>. Online; accessed 15 November 2024.
- [13] Rust Fuzzing Authority. cargo-fuzz: Command line helpers for fuzzing. <https://github.com/rust-fuzz/cargo-fuzz>. Online; accessed 15 November 2024.
- [14] Mohammadreza Ashouri. Rusty: A fuzzing tool for rust. In *Proceedings of the Annual Computer Security Applications Conference*. ACM, 2020.
- [15] Jia Jiang, Hui Xu, and Yangfan Zhou. Rulf: Rust library fuzzing via api dependency graph traversal. In *Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 581–592. IEEE/ACM, 2021.
- [16] Zhiwu Xu, Bohao Wu, Cheng Wen, Bin Zhang, Shengchao Qin, and Mengda He. Rpg: Rust library fuzzing with pool-based fuzz target generation and generic support. In *Proceedings of the 46th International Conference on Software Engineering (ICSE 2024)*, Lisbon, Portugal, April 2024. IEEE.
- [17] LLVM Project. Llvloop terminology. <https://llvm.org/docs/LoopTerminology.html>. Online; accessed 15 November 2024.
- [18] LLVM Project. Llvloop simplify: Canonicalize natural loops. <https://llvm.org/docs/Passes.html#passes-loop-simplify>. Online; accessed 15 November 2024.
- [19] Thomas Lengauer and Robert E. Tarjan. A fast algorithm for finding dominators in a flowgraph. In *ACM Transactions on Programming Languages and Systems*, volume 1, pages 121–141. ACM, 1979.
- [20] Paul Havlak. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(4):557–567, 1997.

- [21] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley. Optimizing seed selection for fuzzing. In *Proceedings of USENIX Security Symposium (USENIX Sec'14)*, pages 861–875, Berkeley, CA, USA, 2014. USENIX Association.
- [22] The Servo Project Developers. Rust url - a url parser for rust. <https://github.com/servo/rust-url>. Accessed: 15 November 2024.
- [23] Chronotope Project Developers. Chrono - a date and time library for rust. <https://github.com/chronotope/chrono/tree/main>. Accessed: 2024-11-15.
- [24] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*, pages 2123–2138, New York, NY, USA, 2018. Association for Computing Machinery.
- [25] RustSec. Rustsec: A security advisory database for rust. <https://rustsec.org/>. Accessed: 15 November 2024.
- [26] Addison Crump, Dongjia Zhang, Syeda Mahnur Asif, Dominik Maier, Andrea Fioraldi, Thorsten Holz, and Davide Balzarotti. Crabsandwich: Fuzzing rust with rust.
- [27] Andrea Fioraldi, Dominik Maier, Dongjia Zhang, and Davide Balzarotti. Libafl: A framework to build modular and reusable fuzzers. In *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS '22)*, Los Angeles, U.S.A., 2022. ACM.
- [28] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. Grey-one: Data flow sensitive fuzzing. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security 2020)*, Boston, MA, 2020. USENIX Association.
- [29] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security 2019)*, Santa Clara, CA, USA, 2019. USENIX Association.
- [30] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. One engine to fuzz 'em all: Generic language processor testing with semantic validation. In *Proceedings of the 2021 IEEE Symposium on Security and Privacy (SP 2021)*, San Francisco, CA, USA, 2021. IEEE.

- [31] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Fuzzing loop optimizations in compilers for c++ and data-parallel languages. In *Proceedings of the ACM on Programming Languages (Proc. ACM Program. Lang.)*, volume 7. ACM, 2023.
- [32] Zhuohua Li, Mingshen Sun, Jincheng Wang, and John C. S. Lui. Mirchecker: Detecting bugs in rust programs via static analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*, pages 1–14, New York, NY, USA, 2021. ACM.