RUDYNA: Towards A Dynamic Analysis Framework for Rust

Shanlin Deng Baojian Hua*
School of Software Engineering, University of Science and Technology of China
Suzhou Institute for Advanced Research, University of Science and Technology of China
dengshanlin@mail.ustc.edu.cn bjhua@ustc.edu.cn

***Abstract*—Rust emerges as a promising safe language and is gaining rapid adoption in security-critical domains. However, Rust programs are susceptible to memory and thread safety issues, making dynamically analyzing Rust issues imperative. Unfortunately, a dynamic analysis framework for Rust is still lacking, hampering the advancement of dynamic analysis for Rust and posing security risks for the language.**

In this paper, we present RUDYNA, the first Rust native dynamic analysis framework to the best of our knowledge. Our framework aims to instrument Rust programs and provide a set of hooks for monitoring runtime events. To this end, we first propose instrumenting the Rust MIR with three rules designed for satisfying its constraints. We then present four instrument strategies to inject various runtime events on MIR. Finally, we develop a hierarchical strategy to instrument configurable hooks to reduce overheads. We implement RUDYNA by extending Rust's official `rustc` compiler and hooks are provided as a Rust library. We conduct evaluation on a set of micro-benchmarks and 4 real-world large Rust projects. Experimental results indicate that RUDYNA preserves the original semantics of programs, with an acceptable runtime overhead ranging from 1.1× to 3.6×, which aligns with built-in instrumentation in `rustc` and similar frameworks for other languages. Moreover, we implement six analyses based on RUDYNA, demonstrating its practical usability.

***Index Terms*—Rust, Dynamic Analysis, Instrumentation**

I. INTRODUCTION

Rust has emerged as a promising safe system programming language, offering strong security guarantees of memory and thread safety with unique characteristics such as ownership mechanism and borrow checkers [1]. These guarantees have led to Rust's rapid adoption in security-critical domains including operating system kernels [2]–[4], browser engines [5], file systems [6], [7], database engines [8], and blockchain protocols [9], [10]. Unfortunately, despite Rust's strong security guarantees, Rust programs remain susceptible to bugs [11]–[15], which may cause serious consequences. For example, the exploit in the Wormhole network caused a loss of more than $320 million [16]. Therefore, detecting bugs in Rust programs is both critical and urgent.

Dynamic program analysis, as a vital technical approach for detecting program bugs, has recently been proposed to address the problem of Rust bug detection [17]–[19]. Unlike static analysis techniques that analyze the target Rust programs [20]–[23] before execution, dynamic analysis executes the target program and monitors the execution process to collect a large spectrum of precise execution information, such as dynamic call graphs and allocations, which are further leveraged to detect memory and API interaction bugs. For example, RUSTSAN [17] instruments unsafe memory allocation points and detects Rust memory vulnerabilities based on shadow memory technology. As another example, both RULF [18] and FRIES [19] employ fuzzing techniques to detect vulnerabilities in Rust library API interactions by generating and executing targeted tests. Leveraging dynamic program analysis, these studies further enhance the security of Rust programs.

Unfortunately, despite these significant progress, dynamic program analysis for Rust remains challenging. We argue that this challenge arises from the fact that a general and Rust native dynamic analysis framework is still lacking. Generally, a dynamic analysis framework refers to a software infrastructure that comprises essential components including program instrumentation, hooks, and analysis algorithms, among others. While prior studies have demonstrated the key value of dynamic analysis frameworks for other languages (*e.g.*, Pin [24] for x86, Jalangi [25] for JavaScript, Wassabi [26] for WebAssembly, and DynaPyt [27] for Python), such a framework for Rust is still lacking. Consequently, Rust developers and security researchers struggle to write dynamic analysis manually that are both labor intensive and error-prone. While one can argue that a potential workaround is to leverage LLVM as a dynamic analysis framework [28] for Rust, unfortunately, doing so will inevitably lose a large amount of Rust-specific semantic information that are critical for the analysis, which undermines the desired effectiveness [17]. Therefore, developing a general and Rust native dynamic analysis framework is an essential first step towards effectively dynamic bug detection for Rust.

In this paper, we present RUDYNA, the *first* Rust native dynamic analysis framework to the best of our knowledge, for promoting the development of dynamic analysis for Rust. Our key idea is to instrument Rust programs and provide a set of hooks for monitoring runtime events, upon which analysis developers can write custom functions based on these hooks to implement corresponding analyses. However, developing such a framework is not trivial and three key challenges should be tackled. **C1:** How to implement instrumentation for Rust's diverse language features? As a high-level program-

* The corresponding author.

TABLE I: Comparison of instrumentation for Rust.

Framework	Target	Automation	Multiple Events	Behavior Manipulation
built-in	LLVM IR	✔	✘	✘
tracing	Rust source	✘	✔	✘
RUDYNA	MIR	✔	✔	✔

ming language, Rust incorporates numerous complex language features. Designing appropriate hooks to instrument these diverse features is a challenge we must confront. Otherwise, the value of this tool is greatly reduced if only a subset of features are supported. **C2:** How to overcome difficulties posed by Rust's unique characters including ownerships and type systems that are absent in other languages? Specifically, to enable Rust native dynamic analysis, we need to insert custom analysis functions into the Rust source code or its intermediate representations. However, if the instrumented code fails to comply with Rust's unique ownership system and type system, instrumentation fails due to compilation errors. **C3:** How to mitigate the runtime performance overhead introduced by instrumentation? Instrumentation potentially introduces a significant number of additional function calls as hooks, leading to considerable compilation overhead and runtime costs. Therefore, minimizing these overheads without sacrificing generality and expressiveness is also challenging.

To address **C1**, we choose to instrument the MIR, the intermediate language in the Rust compiler. By operating on MIR, we can leverage an intermediate representation generated from Rust sources but after desugaring and simplification, thus significantly mitigating the challenge posed by Rust's complex syntax features. Specifically, we propose four instrumentation strategies to selectively instrument the assignment, conditional branches, and function calls, facilitating monitoring of distinct runtime events. To satisfy the Rust constraints to handle **C2**, we select the final phase of MIR optimizations, for the following reasons. First, Rust's ownership system is explicitly established during MIR construction and undergoes core constraint checks afterward. Therefore, instrumenting this final stage avoids stringent checks while preserving rich Rust semantics. Second, inserting analysis hooks into MIR late minimizes interference with MIR optimizations, minimizing performance degradation. Additionally, we design three instrumentation rules to guarantee code validity. To reduce performance overhead discussed in **C3**, we introduce a hierarchical approach to implement configurable hooks. Analysis developers could select events at specific abstraction levels through configuration files to avoid unnecessary performance losses caused by full instrumentation.

To put the contributions of RUDYNA in perspective, Table I compares existing instrumentation approaches for Rust and ours. The built-in instrumentation tool [29] injects LLVM intrinsic functions. However, while this tool is automated, it fails to monitor multiple events. Conversely, the tracing [30] framework can handle multiple events, but does not support behavior manipulation due to its use of Rust sources. On the contrary, our tool RUDYNA operates on the MIR and thus can handle multiple events and manipulate behavior.

We implement a software prototype for RUDYNA, and evaluate it on a set of micro-benchmark containing 16 test cases and 4 real-world and large Rust programs from GitHub, to investigate the semantic fidelity, compilation cost, usability, and runtime overhead. Evaluation results demonstrate that the instrumentation of RUDYNA preserves original program semantics without altering any runtime behavior by default. Moreover, RUDYNA introduces an acceptable compilation time of 6.14% and a binary size increase of 8.18%, while hierarchical and configurable instrumentation can further reduce this compilation overhead. Furthermore, based on RUDYNA, we implement five common dynamic analyses of *BasicBlockCounter*, *ArithmeticChecker*, *BranchConditionFlip*, *CallGraph* and *TraceAll*, one unique analysis of *UnsafeFunctionCounter* which leveraging Rust native features. Each analysis requires only minimal code development or modification, demonstrating its practical usability. Finally, we evaluated the runtime overhead of the designed dynamic analyses. Among them, *TraceAll* introduced a 1.1×-3.6× runtime overhead. Comparisons with built-in instrumentation of `rustc` [29] and dynamic analysis frameworks for other languages [24], [27] indicate this overhead is reasonable and acceptable.

In summary, this paper makes the following contributions:

- We propose the first Rust native dynamic analysis framework RUDYNA.
- We design and implement a software prototype for RUDYNA.
- We conduct systematic evaluations to demonstrate the fidelity and usability of our framework, allowing for analyzing real-world Rust projects with acceptable runtime overhead.
- We make RUDYNA and evaluation data publicly available in the interest of open science to facilitate reproduction, replication, and reuse at `https://doi.org/10.5281/zenodo.16598449`.

The remainder of this paper is organized as follows. Section II introduces the background and motivations. Section III illustrates our design. Section IV presents the experimental evaluation of RUDYNA. Section V discusses limitations and our future work. Section VI reviews related work, and Section VII concludes.

II. BACKGROUND AND MOTIVATION

To be self-contained, in this section we first present the necessary background knowledge on Rust (§ II-A) and dynamic analysis (§ II-B), and then describe the motivation of our work (§ II-C).

A. Rust

Rust [1] is a promising open-source system programming language introduced by Mozilla Research in 2010. Its core design focuses on maintaining high performance while resolving common memory safety issues and concurrency challenges

inherent to C/C++. To this end, Rust incorporates both a unique ownership system [31] and a lifetime system [32] that detect potential common memory safety problems, such as null pointer dereferencing and data races, during compile time. On the other hand, since it does not rely on garbage collection, Rust's safety guarantees achieve zero-cost abstraction, meaning it offers high-level abstractions without compromising runtime efficiency.

Rust employs a multi-layered security verification system to detect vulnerabilities at compile time. When compiling a Rust program, the compiler first transforms the source code into an Abstract Syntax Tree (AST) through lexical analysis, syntax parsing, and macro expansion. The AST undergoes preliminary validation to ensure compliance with language rules. Subsequently, the compiler converts the AST into a High-level Intermediate Representation (HIR) [33] via desugaring, which serves as a critical intermediate layer storing the contents of the current crate being compiled. The HIR is then simplified into a Mid-level Intermediate Representation (MIR) [34], a structure based on control flow graph, which comprises basic blocks, statements, terminators, local variables, locations, rvalues, and operands. Notably, the MIR explicitly encodes ownership semantics and type information. Leveraging these features, the compiler performs borrow checking, Non-Lexical Lifetimes (NLL) [35] analysis, and ownership transfer path verification on the MIR. After multiple optimizations and final validity checks, the MIR is translated into LLVM IR for further compilation.

B. Dynamic Analysis

Dynamic analysis [36] is a widely used technique in software engineering and system security. Unlike static analysis, its core lies in monitoring, evaluating, and interpreting program runtime behavior by executing actual programs, thereby capturing dynamic characteristics that static analysis cannot reveal. Based on these runtime data, dynamic analysis can identify potential performance bottlenecks [37] and security vulnerabilities (*e.g.*, buffer overflows or injection attacks) [38], playing a vital role in maintaining, debugging, and enhancing the security of increasingly complex modern software systems.

Dynamic analysis employs program instrumentation techniques [39], which involves inserting specific control logic (*i.e.*, probes) into the target program. These probes execute during program runtime, capturing internal states and behavioral traces, thereby providing fine-grained data support for dynamic analysis. Currently, program instrumentation primarily falls into three categories: (1) Source-level instrumentation [40], [41] performs source-to-source transformation, replacing target source code with instrumentation code. (2) Intermediate representation level instrumentation [28], [42] inserts custom monitoring code into the program's Abstract Syntax Tree (AST) or intermediate representations. (3) Binary-level instrumentation [43]–[45] dynamically modifies executable files or in-memory processes. These instrumentation techniques effectively capture function call stacks, branch coverage, and other critical data, which provide significant assistance for scenarios like performance profiling, vulnerability detection, and coverage verification.

C. Motivation

Before elaborating on the design of RuDyna, we first illustrate the motivation of this work, which centers on two key considerations.

First, dynamic analysis for Rust remains relatively scarce. Rust's provision of memory safety and thread safety does not equate to Rust programs being devoid of vulnerabilities [11]–[13]. On one hand, Rust introduces the unsafe [46] keyword to retain low-level control capabilities. Code within unsafe blocks can bypass certain compiler checks while performing potentially hazardous operations (*e.g.*, raw pointer manipulation), opening the possibility for potential memory errors. On the other hand, Rust cannot inherently detect logical errors within programs. While there has been extensive research into Rust vulnerability detection using static program analysis, these approaches still suffer from issues such as insufficient accuracy [21], [22]. A common technique to improve accuracy is employing dynamic analysis, which has seen significant advancements in research for other languages [38], [47]. However, based on our investigation, dynamic analysis techniques specifically targeting Rust are still underdeveloped compared to existing research for other languages. Advancing progress in this area will undoubtedly enhance the security of Rust programs.

Second, the absence of Rust native instrumentation compromises the effectiveness of dynamic analysis. The Rust compiler utilizes LLVM as its backend, meaning LLVM-based dynamic analysis tools can be directly applied to instrument Rust programs. Taking the renowned memory vulnerability detection tool AddressSanitizer [47] as an example, this tool detects buffer overflows, use after free, and other vulnerabilities in C/C++ and Rust programs by instrumenting memory allocation and deallocation functions at the LLVM IR. However, existing research [17] has shown that using AddressSanitizer on Rust programs suffers from a performance gap. This is because the tool does not account for Rust's safety semantics, leading to significant redundant instrumentation during detection and consequently causing performance degradation. Based on this observation, we posit that designing a Rust native dynamic analysis framework could enable more effective dynamic analysis tools built upon it.

III. Design

In this section, we present the design of RuDyna. We begin with an overview of the workflow of framework (§ III-A), then detail our approach to instrumentation (§ III-B) and the hierarchy and configurability of runtime events (§ III-C). Finally, we illustrate the implementation of RuDyna (§ III-D).

A. Overview

As a dynamic analysis framework, our key goal is to instrument Rust programs and provide a series of hooks to

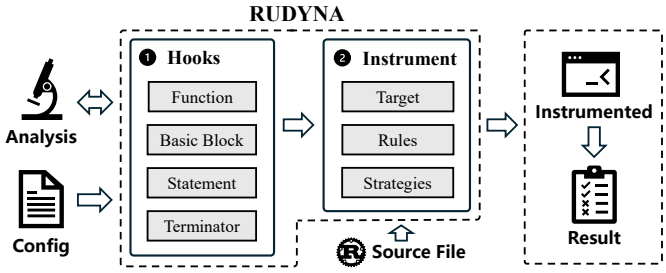

Fig. 1: An overview of RUDYNA's workflow.

monitor the runtime events. Guided by this goal, RUDYNA comprises hooks (❶) and instrument (❷) components. The hooks feature a hierarchical and configurable design, enabling analysis at different levels. The instrument component operates on MIR, injecting hooks into the program according to defined instrument rules and strategies while ensuring compilation correctness and semantic integrity.

To better illustrate the workflow of RUDYNA, we describe how the framework assists developers in implementing custom dynamic analysis in Fig. 1. First, developers can select runtime events at desired levels through config file and disable irrelevant instrumentation to avoid unnecessary runtime overhead. Subsequently, with the hooks provided by RUDYNA, users could implement their analysis procedures conveniently. Then the instrumenter injects the corresponding instrumentation code into the MIR of the program under analysis and outputs an instrumented Rust binary. Finally, the instrumented program performs users' analysis through inserted hooks and obtains results during execution.

B. Instrument for Rust

In this section, we describe the instrumentation performed by RUDYNA by illustrating instrument target to insert hooks, the instrument rules complying the strict type system in Rust and several instrument strategies.

Instrument Target. The instrument target refers to the program representation at which we perform instrumentation and the stage during which hooks are added to the program. Determining the instrument target effectively guides the subsequent formulation of instrument rules and the selection of instrument strategies.

The design goal of Rust native requires that the instrument target should contain sufficient semantic information of Rust, which means we need to select among Rust source language, abstract syntax tree (AST), HIR and MIR. We choose to perform the instrumentation on MIR based on two reasons: (1) MIR is an intermediate representation obtained after desugaring and simplifying Rust programs. During the generation of MIR, Rust's complex syntax is transformed into simple MIR data structures, and the ownership mechanism is explicitly established, which significantly reduces the complexity of program instrumentation. (2) MIR is associated with HIR closely, which allows us to obtain rich semantic information of Rust in the MIR. Specifically, we position the instrumentation process at the final stage of MIR optimization, for Rust's core borrow checking (`rustc_borrowck`) and Non-Lexical Lifetimes (NLL) checking are performed after the construction of MIR. This approach simplifies Rust checks by operating at the end of optimization while preventing instrumentation from affecting internal optimizations (*e.g.*, dead code elimination) in Rust.

Algorithm 1: Instrument on MIR.

Input: $\mathbb{F}$: a function body
Output: $\mathbb{F}$: the instrumented function body

```
1  Function InstrumentMIR(𝔽):
2      NewBlocks = [ ];
3      for each block 𝔹 in 𝔽 do
4          CurrBlock = 𝔹;
5          for each statement s in CurrBlock do
6              if s needs to instrument then
7                  (first, second) = split_at(s + 1);
8                  𝔹′ = new_block();
9                  𝔹′.Stmts = second;
10                 𝔹′.Term = CurrBlock.Term;
11                 NewBlocks.add(𝔹′);
12                 CurrBlock.Stmts = first;
13                 CurrBlock.Term = get_hook(s);
14                 CurrBlock.Stmts.remove_last();
15                 CurrBlock = 𝔹′;
16         Instrument CurrBlock.Term;
17     append(𝔽.Blocks, NewBlocks);
18     return 𝔽;
```

Instrument Rules. Instrument rules describe the conditions that must be met when inserting hooks into a Rust program. These conditions ensure that the inserted functions satisfy the ownership, lifetime and type constraints of Rust to avoid compilation errors caused by instrumentation. Since we determine to perform instrumentation at the final stage of MIR optimization to avoid strict constraints in Rust, injecting arbitrary code may still cause crashes for violating the constraints in MIR. To obey the constraints of MIR, we mainly design the following three rules.

Rule 1: Parameter types of instrumentation functions must match. As we replace operations in Rust with hooks in the form of function calls, the parameter types of these hooks must match those of the corresponding operations according to type checking. For instance, to instrument the common Rust arithmetic operation `_2 = Sub(copy _1, const 1_i32)` for $i32$ type, we have to replace the subtraction operation with hook `my_sub` while the corresponding parameters must be of type $i32$.

Rule 2: Avoid reusing Move-type Operands. During MIR construction, Rust's ownership system is explicitly built, where operands are wrapped as `Move(Place<'tcx>)`, `Copy(Place<'tcx>)`, and Constant types. The `Copy` type corresponds to Rust's copy semantics, indicating value cre-

TABLE II: Instrument strategies in RUDYNA

Id	Original MIR	Instrumented MIR
1	`v1 = bop(v2,v3)`	`v1 = _type_bop(v2,v3)`
2	`v1 = uop(v2)`	`v1 = _type_uop(v2)`
3	`v1 = cmp(v2,v3)`	`v1 = _type_cmp(v2,v3)`
4	`SwitchInt(v,...)`	`v' = _bool_use(v)` `SwitchInt(v',...)`
5	`v1 = Call(...)`	`_call(caller,callee)` `v1 = Call(...)`

6	`v1 = Call(v2,v3,...)`	`v2' = _type_use(v2)`
		`v3' = _type_use(v3)`
		`v1' = Call(v2',v3',...)`
		`v1 = _type_use(v1')`

ation by loading from a specified location. The `Move` type corresponds to Rust's ownership transfer semantics, similarly creating a value by loading from a location, but the compiler may overwrite that location with uninitialized bytes afterward. Any attempt to reload this location would violate MIR invariants. Under this rule, code inserted by RUDYNA does not modify existing operand types, while newly added operands are exclusively used within the inserted code with strictly defined types.

Rule 3: The instrumentation process must maintain valid MIR structure. Since MIR defines function calls as Terminators to handle runtime panics, preserving legal MIR structure necessitates basic block splitting for RUDYNA, thus motivating our instrumentation algorithm design in Alg. 1. The algorithm iterates through each basic block `B` and every statement `s` within a function. If `s` is an operation requiring instrumentation, the algorithm creates a new basic block `B'`, moves all statements and the terminator following `s` verbatim into `B'`. It then modifies the terminator of `B` to the corresponding hook, and sets its successor as `B'`. Finally, the algorithm continues traversing statements in `B'`. After processing all statements in the basic block, RUDYNA will instrument the terminator. Instrumentation for terminator resembles that for statement, but requires additional codes, which will be elaborated in subsequent instrument strategies. Moreover, the algorithm employs an additional array to record newly created basic blocks, inserting this array after the original basic block array post-instrumentation. This constraint arises because jump relationships in MIR rely on array indices, and inserting new blocks into the original array could disrupt existing jump relations.

Instrument Strategies. RUDYNA employs a set of instrumentation strategies to substitute operations in Rust programs with corresponding hooks, enabling systematic detection and manipulation of runtime events. As demonstrated in Table II, these implemented strategies currently represent our core instrumentation mechanisms, with minor simplifications to ease the presentation (*e.g.*, type omitted), while the instrumentation of additional Rust operations will be addressed in future work. The following discusses in more detail the instrument strategies we employed.

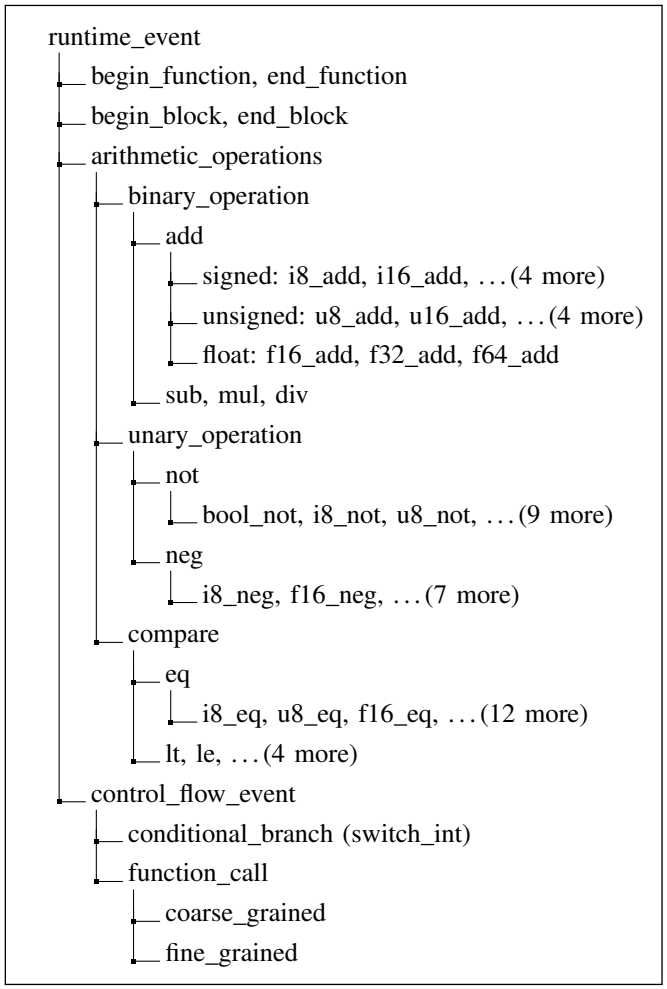

Fig. 2: Hierarchy of hooks in RUDYNA

Assignment statements. As shown in rows 1, 2, and 3 of Table II, we primarily consider three types of assignments: `binop`, `unary` and `compare`, to provide a means of monitoring arithmetic operations in a program. RUDYNA adopts a substitution approach, replacing assignment statements with corresponding hooks to implement code instrumentation. Note that the selected hooks must match the types of the original operands to satisfy type requirements.

Conditional branches. Conditional branches are a critical component of control flow in Rust programs, making the monitoring and potential modification of branch conditions a key objective for dynamic analysis frameworks. To this end, as illustrated by row 4, RUDYNA first extracts the branch condition variable `v` and hooks to intercept it. Through hook parameters, users can access the variable and optionally supply a return value to modify the condition. The branch then executes the conditional jump based on the new condition `v'` provided by the hook.

Function calls. Monitoring function calls is an essential component of dynamic analysis frameworks [25], [27]. However, designing a unified approach to monitor or control

```rust
use std::collections::HashMap;
use std::sync::Mutex;
static COUNTER: std::sync::LazyLock<Mutex<
    HashMap<(u32, u32), u32>>> = std::sync::
    LazyLock::new(|| Mutex::new(HashMap::
    default()));

pub fn _instrument_block_begin(func: u32, bb:
    u32) {
  let mut counter = COUNTER.lock().unwrap();
  let key = (func, bb);
  let entry = counter.entry(key).or_default();
  *entry += 1;
  drop(counter);
}
```

Fig. 3: Example analysis for basic block counting.

```rust
pub fn _i32_arithmetic(left: i32, right: i32,
    kind: u8) -> i32 {
  let widen_left = left as i128;
  let widen_right = right as i128;
  let kind = kind.try_into().expect("...");
  let result = match kind {
    OpKind::Add => widen_left + widen_right,
    OpKind::Sub => widen_left - widen_right,
    OpKind::Mul => widen_left * widen_right,
    OpKind::Div => widen_left / widen_right,
    _ => unreachable!(),
  };
  if result > i32::MAX as i128 || result < i32::
      MIN as i128 {
    // handle overflow
  }
  result as i32
}
```

Fig. 4: Example analysis to check arithmetic overflow of $i32$.

```rust
use std::random::random;
pub fn _flip_if_cond(cond: bool) -> bool {
  if random() { !cond } else { cond }
}
```

Fig. 5: Example analysis to manipulate conditional branch.

function calls on MIR remains challenging, because Rust function calls often contain differing numbers and types of parameters. To monitor every argument and return value of a specific function call, the instrumentation function must possess the exact same number and types of parameters as that call, which also means it cannot be used to monitor other calls with different parameters. Conversely, to achieve generic instrumentation for function calls, the function's parameters and return value cannot be accounted for, which diminishes its utility. To address this, we developed two distinct function call instrumentation strategies, offering varying levels of manipulation capabilities to balance ease of use with functional flexibility. The first approach (row 5) offers coarse-grained monitoring of function calls, capturing only inter-procedural jump relationships. While it enables convenient instrumentation for all functions with lower runtime overhead, this method suits analyses unconcerned with execution details (*e.g.*, dynamic call graph construction). The second approach (row 6) provides fine-grained manipulation capabilities for function calls. Users implement custom analyses for specific functions or function categories, and RUDYNA instruments designated calls by hooking into parameters and return values based on user configurations.

Selective instrument. In Rust projects that rely on numerous external libraries, instrumenting all dependencies may trigger cascading effects leading to unacceptable performance degradation. To mitigate this, RUDYNA avoids unnecessary runtime costs by selectively instrumenting only user-specified crates of interest.

C. Hierarchy and Configurability of Hooks

A key design of RUDYNA is to organize analysis hooks into a hierarchical structure of runtime events, with instrumentation for specific events being configurable. This design serves twofold. Firstly, hierarchical and configurable events provide analysis developers with more precise monitoring of runtime events while reducing both the overhead and developer burden. Secondly, the hierarchical design abstracts away underlying complexity. For instance, to satisfy Rust's type requirements, instrumentation for addition operations must be implemented separately for each type (*e.g.*, $i8$, $i16$, *etc.*). If a user wants to instrument all addition operations without concern for the specific type, a non-hierarchical structure would require them to write instrumentation for every possible type, significantly diminishing RUDYNA's utility.

Based on this design, RUDYNA does not provide a set of analysis hooks at a fixed granularity level, but organizes these hooks into several distinct abstraction layers, and all these layers and events can be fully configured through the configuration. As shown in Fig. 2, the root node of this structure is `runtime_event`, which instruments the entire program. Below the root node at the second level are function operations, basic block operations, arithmetic operations, and control flow operations. Function operations include `begin_function` and `end_function`, which operate on the entry and exit of each function, respectively. Similarly, basic block operations include `begin_block` and `end_block`, operating on the entry and exit of each basic block. Arithmetic operations are further categorized into binary operations, unary operations, and comparison operations, subdivided based on operation type and operand types. Finally, control flow operations focus on conditional branches and function calls, as we have described before.

To illustrate how hierarchical hooks aid users in monitoring different event types, we present several examples of analysis based on RUDYNA as follows.

BasicBlockCounter. Counting basic block via instrumentation is a fundamental technique for program analysis and optimization and its results provide critical data support for performance optimization, code coverage testing, and program behavior analysis [24], [48]. As shown in Fig. 3, with the basic block operation `begin_block` provided by RUDYNA, users can increment a counter at the entry of each basic block and

obtain the execution count of the corresponding basic block during runtime.

ArithmeticChecker. Arithmetic overflow is a common programming error [49]. In Rust, integer arithmetic operations in debug mode are replaced with overflow-checked versions. If an arithmetic overflow occurs during runtime, the overflow check aborts the program execution, which contributes to Rust's strong safety guarantees. As shown in Fig. 4, we implement an analysis similar to Rust's built-in overflow checking using RUDYNA for $i32$ arithmetic operations. Based on hook `_i32_arithmetic`, the analysis captures the operation type and operands, internally determining whether the calculation results in overflow. Notably, this hook allows modifying the computation results, enabling greater manipulation capabilities than the built-in overflow checks.

BranchConditionFlip. Flipping branch conditions is a critical method for dynamic analysis frameworks to manipulate control flow. Fig. 5 illustrates how RUDYNA utilizes the `_flip_if_cond` hook to randomly alter branch conditions. In practical applications, users can design more complex flipping conditions, such as implementing concolic execution [50], [51] or enforced execution [52], [53].

D. Prototype Implementation

To validate our design, we implement a software prototype for RUDYNA, consisting a component of instrument and a series of hooks. We integrate the instrumenter as an additional optimization pass at the final stage of MIR optimization on top of the latest Rust compiler (`rustc` 1.90.0-nightly). This design facilitates accessing compilation information and performing accurate, convenient MIR manipulation. Correspondingly, we implement a frontend compiler flag in `rustc` to control the instrument's activation. The option remains disabled by default to avoid impacting ordinary compilation workflows. The hooks are implemented as a Rust library with a hierarchical design which mentioned before, allowing compile-time hook information retrieval and instrumentation. In order to use RUDYNA, users only need to add the library as a dependency in their project's `Cargo.toml`, import it in the root module (without explicitly using its functions), and enable the instrumentation option, thereby achieving rapid instrumentation for Rust programs. Finally, we distribute the prototype in our open source.

IV. EVALUATION

To understand the effectiveness of RUDYNA, we evaluate it on micro-benchmarks and real-world Rust programs. Specifically, our evaluation aims to answer the following questions:

RQ1: Semantic Fidelity. Does the instrumentation of RUDYNA remain faithful to the original program's semantics?

RQ2: Compilation Overhead. How much time overhead does the program instrumentation of RUDYNA introduce during compilation, and what is the binary size inflation?

RQ3: Framework Usability. How complex is it to design dynamic analysis based on RUDYNA?

RQ4: Runtime Overhead. How much runtime overhead does using RUDYNA for dynamic analysis introduce?

All the experiments and measurements are performed on a server with one 12 physical Intel i7 core (20 hyperthread) CPU and 128 GB of RAM. The machine runs 64-bit Ubuntu 24.04 Linux with kernel version 6.8.0. The Rust programs are compiled with rustc version 1.90.0-nightly build.

A. Datasets

We conduct the evaluation using two datasets: (1) a set of micro-benchmarks, consisting of 16 test cases; (2) a set of real-world benchmarks, comprising 4 open source Rust projects.

Micro-benchmarks. Given the complexity of real-world projects, constructing micro-benchmarks for initial correctness and functional validation of the framework is crucial. To ensure the reliability and comprehensiveness of the verification implementation, test cases in the suite should have deterministic outputs for given inputs, while covering diverse functionalities and remaining as simple as possible. To this end, we adopt a series of open-source algorithms written in Rust from GitHub, which span different domains, ranging from simple sorting and searching to complex data structures and machine learning. Among those algorithms, we select one representative implementation from each domain, ultimately resulting in the 16 small test cases shown in rows 1-16 of Table III.

Real-world projects. We further evaluate RUDYNA using real-world Rust projects, and our selection is guided by three principles. First, the projects should be open-source and widely adopted to thoroughly validate the framework's usability and impact. To this end, we select projects written in Rust from GitHub, measuring their popularity through their number of stars. Second, the projects should be actively maintained. Given the rapid evolution of the Rust language, its internal features have undergone changes in recent years [54], rendering many archived projects incompatible with the latest `rustc` compiler. Since RUDYNA does not target compatibility with older compiler versions, these outdated projects pose significant challenges for our evaluation. Actively maintained projects, however, are often compatible with the latest Rust versions, facilitating our experiments. Finally, we prioritize projects containing regression tests, as verifying the framework's semantic fidelity and performance through regression tests is a primary method in our experimental evaluation.

As a result, we select four projects from different domains: rust-bitcoin, RustPython, shadowsocks-rust, and candle, as presented in rows 17-20 of Table III. Among these, rust-bitcoin is a library with support for de/serialization, parsing and executing on data-structures and network messages related to Bitcoin. RustPython is a Python interpreter written in Rust, while shadowsocks-rust is a Rust port of shadowsocks. Candle is a minimalist ML framework for Rust with a focus on performance and ease of use.

B. RQ1: Semantic Fidelity

To answer RQ1, we apply RUDYNA to both micro-benchmarks and real-world projects for comprehensive instru-

TABLE III: Compilation overhead and consistency of RUDYNA on datasets.

#	Description	CT (s), BI / AI	CT Overhead	Size (MB), BI / AI	Size Overhead	Consistency
1	n queens	0.271 / 0.289	6.64%	1.75 / 1.88	7.43%	✔
2	fast factorial	2.078 / 2.079	0.05%	7.02 / 7.03	0.14%	✔
3	counting bits	0.230 / 0.231	0.43%	1.11 / 1.23	10.81%	✔
4	base64 encode	0.273 / 0.277	1.47%	1.59 / 1.73	8.81%	✔
5	run length encode	0.280 / 0.298	6.43%	1.81 / 1.94	7.18%	✔
6	binary to decimal	0.805 / 0.825	2.48%	2.01 / 2.13	5.97%	✔
7	floyds algorithm	0.267 / 0.273	2.25%	1.26 / 1.43	13.49%	✔
8	present value	0.260 / 0.269	3.46%	1.40 / 1.54	10.0%	✔
9	huffman encode	0.340 / 0.347	2.06%	2.92 / 3.09	5.82%	✔
10	closest points	0.311 / 0.321	3.22%	2.24 / 2.40	7.14%	✔
11	dijkstra algorithm	0.715 / 0.740	3.50%	12.41 / 12.59	1.45%	✔
12	stable matching	0.370 / 0.390	5.41%	4.00 / 4.19	4.75%	✔
13	k means	1.901 / 1.948	2.47%	4.40 / 4.55	3.41%	✔
14	miller rabin	1.666 / 1.714	2.88%	3.27 / 3.47	6.12%	✔
15	fibonacci search	0.238 / 0.245	2.94%	1.15 / 1.30	13.04%	✔
16	quick sort	1.911 / 1.937	1.36%	4.63 / 4.81	3.89%	✔
17	rust_bitcoin	10.230 / 10.803	5.60%	63.1 / 69.4	9.98%	✘
18	RustPython	42.280 / 46.303	9.52%	663.84 / 761.63	14.73%	✔
19	shadowsocks-rust	41.737 / 43.153	3.39%	1690.01 / 1724.39	2.03%	✔
20	candle	47.835 / 50.732	6.06%	1434.45 / 1519.85	5.95%	✔

AI: After Instrumentation; BI: Before Instrumentation; CT: Compilation Time.

TABLE IV: Consistency of RUDYNA on real-world projects.

Projects	Passed	Failed	Pass Rate
rust_bitcoin	403	1	99.75%
RustPython	183	0	100%
shadowsocks-rust	11	0	100%
candle	221	0	100%

mentation, ensuring that the hooks do not alter any program behaviors. For micro-benchmarks, we design specific inputs and verify the consistency of the output results. For real-world projects, we execute the built-in tests using command `cargo test` and checked whether the instrumented programs could pass all tests.

The experimental results are presented in last column (i.e, Consistency) of Table III. All micro-benchmark cases produce identical results both before and after instrumentation. For the four real-world projects, all pass their tests after instrumentation except for rust-bitcoin. As illustrated in Table IV, the rust-bitcoin project runs a total of 404 tests, which the instrumented program passes 403 of them, failing only one test, resulting in a pass rate of 99.75%. Examining the output logs reveals that this specific test is expected to panic, but the instrumented program runs normally instead, causing the test failure.

To investigate the cause of this semantic discrepancy, we manually inspect the test's source code. The inspection confirms that the test case is designed to panic by a $u64$ arithmetic overflow. Our instrumentation process has disabled Rust's overflow checks for arithmetic operations, leading to the observed difference. Subsequently, we re-enable overflow checking and re-run the tests. As expected, the instrumented program then passes all tests successfully.

This experiments show RUDYNA possesses excellent semantic fidelity, and the inserted hooks do not, by default, alter the program's execution results.

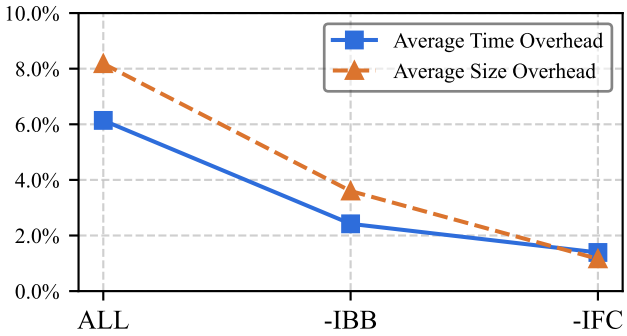

Fig. 6: Compilation overhead under different configurations.

C. RQ2: Compilation Overhead

To answer RQ2 by investigating the overhead during the compilation, we apply RUDYNA to both micro-benchmarks and real-world projects, and measure the compilation time and binary size.

As shown in column 3-6 of Table III, RUDYNA introduces both negligible compilation overhead (2.94%) and little binary size bloat (6.84%) to micro-benchmarks. For the real-world projects, compilation time increases by an average of 6.14%, and binary size grows by an average of 8.18%. Overall, RUDYNA's compilation overhead remains acceptable.

Furthermore, as RUDYNA performs instrumentation on basic blocks and function calls quite frequently, we hypothesize that these two components significantly impact compilation overhead. To verify this, leveraging our hierarchical design, we sequentially disable instrumentation for these operations via configuration file. The experimental results are presented in Fig. 6. Eliminating instrumentation for basic block (-IBB) reduces RUDYNA's compilation time and binary size bloat

TABLE V: Example analyses written on top of RUDYNA.

Analysis	Description	LoC
BasicBlockCounter	Counts basic blocks executed	11
ArithmeticChecker	Checks the arithmetic operation	16
BranchConditionFlip	Flips branch condition randomly	4
UnsafeFunctionCounter	Counts unsafe functions executed	8
CallGraph	Computes a dynamic call graph	18
TraceAll	Instruments all supported events	5

LoC: Lines of Code.

by 3.72% and 4.58%, respectively, and disabling function call instrumentation (-IFC) further reduces these costs by an additional 1.03% and 2.44%. These experiments demonstrate that hierarchical, configurable instrumentation is effective in lowering compilation costs.

D. RQ3: Framework Usability

As a general dynamic analysis framework, RUDYNA must possess excellent usability to reduce the design difficulty of dynamic analysis for developers. In previous section, we have designed three example analyses: *BasicBlockCounter*, *ArithmeticChecker*, and *BranchConditionFlip*. To further verify RUDYNA's usability, we implemented three additional analyses.

UnsafeFunctionCounter. In Rust, unsafe code has become the primary source of vulnerabilities due to bypassing certain safety checks [23]. The detection and remediation of unsafe vulnerabilities now constitute a critical area of Rust security research. To demonstrate the role of Rust native feature in this domain, we implement an unsafe function counter analysis based on RUDYNA. Leveraging the `begin_function` hook, we identify unsafe functions through the hook's provided parameter and tally their occurrences. While this analysis is relatively simple, the hook's ability to distinguish between safe and unsafe functions offers a crucial differentiation capability, enabling developers to build more sophisticated detection logic upon this foundation.

CallGraph. In the domain of program analysis and optimization, call graphs serve as critical representations of dynamic relationships during program execution, playing a pivotal role in understanding system behavior [55]. Based on RUDYNA, we employs coarse-grained function call instrumentation to record caller and callee information at call sites, thereby constructing a dynamic call graph.

TraceAll. Similar to our approach in RQ1 and RQ2, this analysis inserts all hooks included in RUDYNA into the program without any operations. It represents an extreme scenario where complete routine instrumentation is implemented to monitor all supported events during program execution. While offering no specific utility, it reveals the maximum runtime performance penalty that RUDYNA could theoretically impose on programs.

The Table V enumerates the code lines required to implement the aforementioned analyses. The results show that implementing these analyses demands only minimal code, demonstrating the usability of RUDYNA.

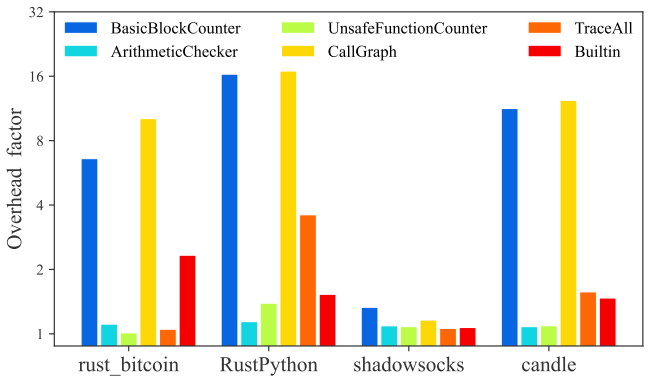

Fig. 7: Runtime overhead of different analyses.

E. RQ4: Runtime Overhead

To evaluate the runtime performance overhead imposed by RUDYNA, we conduct the example analyses (excluding *BranchConditionFlip*) and compare the execution time of programs before and after instrumentation. *BranchConditionFlip* is excluded because it deliberately alters program control flow; thus its execution time is meaningless for comparison.

Fig. 7 presents the evaluation results, where each bar represents the overhead factor relative to the execution time of the uninstrumented program. The experimental results show that *TraceAll* analysis incurs an execution time overhead ranging from 1.1× to 3.6×, with an average of 1.8×. As *TraceAll* represents the worst-case scenario for RUDYNA, we consider its current performance penalty acceptable.

Meanwhile, *ArithmeticChecker* and *UnsafeFunctionCounter* introduce lower overheads of 1.1× and 1.14× respectively. By avoiding unnecessary instrumentation, these two analyses demonstrate even smaller performance impacts than *TraceAll* on the RustPython and candle, further validating the importance of hierarchical and configurable instrumentation for reducing runtime costs in dynamic analysis.

Notably, *BasicBlockCounter* (1.4×-16.5×) and *CallGraph* (1.2×-17×) consistently exhibit significant overhead across almost all test cases. We attribute this to two factors: (1) The inherent abundance of basic blocks and function calls in programs inevitably amplifies the performance penalty through frequent instrumentation-triggered operations (a pattern also observed in *TraceAll*); (2) Analysis cost depends not only on RUDYNA but also on the computational complexity of the analysis itself. Our example analyses involving file I/O operations and global state management through Mutex in Rust, which contribute substantially to this overhead.

Furthermore, all analyses introduce minimal performance impact in the shadowsocks-rust. This is because the project implements shadowsocks ports mainly in Rust, whereas the core system itself is not primarily written in Rust. Since RUDYNA only instruments Rust code, it consequently exhibits lower overhead.

Finally, we conduct a comparison based on *TraceAll* with built-in instrumentation in Rust and frameworks for other lan-

guages. We do not evaluate the tracing crate, as implementing it would necessitate substantial modifications to the source code of those projects, which is impractical. The results of built-in instrumentation are displayed as *Builtin* in Fig. 7. And experimental results show that RUDYNA outperforms built-in on rust_bitcoin but underperforms it in RustPython. On the other two projects, both exhibit similar performance. However, built-in instrumentation is used for profiling, which can monitor fewer events and cannot modify program behavior. Therefore, we believe RUDYNA offers greater capabilities while achieving performance close to built-in. We then use *TraceAll* for a indirect comparison with framework for other languages, while a direct comparison is impossible. In DynaPyt, the cost of *TraceAll* ranges from 1.2× to 16× [27]. In Pin, after final optimizations, integer instrumentation costs 2.5× while floating-point instrumentation costs 1.4× [24]. Based on the comparison, we believe the runtime overhead introduced by RUDYNA (1.1×-3.6×) is reasonable and acceptable.

V. DISCUSSION

In this section, we discuss some possible enhancements to this work, along with directions for future work.

First, the coverage of hooks is still inadequate. Our current design primarily covers Rust's arithmetic operations, branch switching, and function calls. These designs address only part of the Rust MIR, but fail to cover more complex structures such as `SetDiscriminant`, `Ref`, and `RawPtr`. While existing methods can instrument these structures, how to design appropriate hooks for developer convenience requires further exploration.

Second, the design of hooks remains insufficient. Existing hooks only provide necessary parameters related to their corresponding operations. However, during our extension of `rustc`, we discover that hooks could offer richer Rust information (*e.g.*, unsafe attributes). These additional details could significantly enhance the framework's functionality, but may incur performance penalties. Therefore, we need to further evaluate and adjust the hook design.

Third, the implementation of RUDYNA can be further improved. Constrained by Rust's type system, we need distinct hooks for the same operation across different parameter types. Although we conceal some complexity from developers through library module design, this is not a fundamental solution. To address this limitation, we are attempting to refactor the implementation using generics.

Finally, while we have designed six example analyses within the framework to demonstrate RUDYNA's usability, these analyses are relatively simplistic and insufficient for detecting real-world vulnerabilities in Rust programs. To address this limitation and explore RUDYNA's potential, a critical future direction involves developing a dynamic analysis for Rust safety detection under the framework.

VI. RELATED WORK

In recent years, substantial research has been conducted on Rust and dynamic analysis frameworks most relevant to our work.

Static Analysis for Rust. Static program analysis [56] is a crucial technique for program optimization and vulnerability detection that has been widely applied in Rust research. For instance, Li et al. [21] designed MirChecker, a fully automated vulnerability detection tool for Rust, based on abstract interpretation and a monotonic framework. Bae et al. [22] proposed three significant unsafe error patterns and identified real vulnerabilities in Rust packages through pattern matching. Similarly, Qin et al. [23] conducted a comprehensive empirical study on Rust security issues, designing and implementing five static detectors. These detectors have pinpointed a large number of potential vulnerabilities while maintaining a low false positive rate. Beyond these, static program analysis-based research also includes Yuga [57], which detects Rust lifetime errors, and VRust [16], which detects vulnerabilities in the Solana blockchain. These static detection methods have made significant progress and contributed substantially to Rust security.

Dynamic Analysis for Rust. Dynamic program analysis is an effective approach for addressing issues related to software correctness, security, and performance. Techniques in this domain are also being applied in the Rust ecosystem. For instance, leveraging Rust's safety mechanisms, Cho et al. [17] optimized AddressSanitizer to propose RustSan, achieving more efficient dynamic detection of memory safety issues in Rust. As another form of dynamic analysis, Jiang et al. [18] designed RULF, a tool that utilizes API dependency graphs to fuzz Rust libraries for detecting library bugs. Building upon this research, Yin et al. [19] further introduced ecosystem-guided target generation techniques to enable efficient Rust library testing. These studies, from another perspective, further enhance the security of Rust. However, compared to static analysis, research on dynamic analysis for Rust remains relatively understudied and warrants further future exploration.

Instrumentation on Rust. Instrumentation is a common technique in dynamic analysis. Currently, tools designed for Rust instrumentation include the built-in instrumentation within `rustc` and the tracing crate. To support profile-guided optimization (PGO), `rustc` includes a built-in instrumentation mechanism [29] that automatically inserts LLVM intrinsic functions (`llvm.instrprof.increment`) at function calls and branches to enable branch coverage tracking. However, this approach lacks support for other event types and cannot modify program behavior. In contrast, tracing [30] is a source-level instrumentation framework for Rust, primarily used for collecting structured diagnostic information about events. While instrumentation with tracing provides broad monitoring capabilities for Rust events, it requires developers to manually annotate their programs using functions and macros provided by the crate, which imposes an additional burden.

Dynamic Analysis Frameworks. Existing dynamic analysis frameworks for other programming languages provide multi-level program analysis spanning from source code to binary. For instance, DynaPyt [27] for Python and Jalangi [25] for JavaScript both implement code instrumentation through

source-to-source transformation. Although significantly different from our objectives and design, the former inspires our hierarchical instrumentation strategy, and the latter guides certain aspects of our instrumentation approach. Frameworks like DiSL [58] and RoadRunner [59] for Java demonstrate dynamic analysis implementations through Java bytecode instrumentation. Additionally, tools such as DynamoRIO [60], Pin [24], and Valgrind [61] target instrumentation for x86 binaries, while Wassabi [26] instruments the WebAssembly. These frameworks operating at different abstraction levels have significantly advanced dynamic analysis research. Building on these foundations, this paper presents the first Rust native dynamic analysis framework.

VII. Conclusion

In this work, we present the first Rust native dynamic analysis framework RuDyna. The framework instruments Rust programs in order to provide developers with a series of hooks for dynamic analysis. To this end, we first select the Rust MIR for instrumentation, which preserves rich Rust semantic information while bypassing the complexity of ownership and lifetime checks. We then design three instrumentation rules to ensure correctness on MIR and propose three strategies to instrument arithmetic operations, function calls, and conditional branches. Moreover, we implement hierarchical and configurable hooks that balance usability against minimized runtime overhead. We implement a software prototype for RuDyna and conduct experiments to evaluate its semantic fidelity, usability and overhead both in compilation and runtime. The experimental results demonstrate that RuDyna achieves excellent semantic fidelity, strong usability, and acceptable compilation and runtime overhead. Overall, this work contributes positively to the advancement of dynamic analysis research for Rust.

References

[1] "The rust programming language - the rust programming language," https://doc.rust-lang.org/stable/book/.

[2] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis, "Multiprogramming a 64kb computer safely and efficiently," in *Proceedings of the 26th Symposium on Operating Systems Principles*. Shanghai China: ACM, Oct. 2017, pp. 234–251.

[3] "Redox - your next(gen) os - redox - your next(gen) os," https://www.redox-os.org/.

[4] "Rust for linux," https://github.com/Rust-for-Linux.

[5] T. S. P. Developers, "Servo aims to empower developers with a lightweight, high-performance alternative for embedding web technologies in applications." https://servo.org/.

[6] "Redox-os/tfs," Redox OS, Feb. 2025.

[7] S. Miller, K. Zhang, M. Chen, and R. Jennings, "High velocity kernel file systems with bento."

[8] "Tikv/tikv," TiKV Project, Mar. 2025.

[9] "Diem/diem: Diem's mission is to build a trusted and innovative financial network that empowers people and businesses around the world." https://github.com/diem/diem.

[10] "Openethereum/parity-ethereum: The fast, light, and robust client for ethereum-like networks." https://github.com/openethereum/parity-ethereum.

[11] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamarić, and L. Ryzhyk, "System programming in rust: Beyond safety," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. Whistler BC Canada: ACM, May 2017, pp. 156–161.

[12] Z. Yu, L. Song, and Y. Zhang, "Fearless concurrency? understanding concurrent programming safety in real-world rust software," 2019.

[13] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang, "Understanding memory and thread safety practices and issues in real-world rust programs," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. London UK: ACM, Jun. 2020, pp. 763–779.

[14] H. Xu, Z. Chen, M. Sun, Y. Zhou, and M. R. Lyu, "Memory-safety challenge considered solved? an in-depth study with all rust cves," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 1, pp. 1–25, Jan. 2022.

[15] M. Cui, C. Chen, H. Xu, and Y. Zhou, "Safedrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, pp. 1–21, Oct. 2023.

[16] S. Cui, G. Zhao, Y. Gao, T. Tavu, and J. Huang, "Vrust: Automated vulnerability detection for solana smart contracts," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. Los Angeles CA USA: ACM, Nov. 2022, pp. 639–652.

[17] K. Cho, J. Kim, K. D. Duy, H. Lim, and H. Lee, "Rustsan: Retrofitting addresssanitizer for efficient sanitization of rust."

[18] J. Jiang, H. Xu, and Y. Zhou, "Rulf: Rust library fuzzing via api dependency graph traversal," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Melbourne, Australia: IEEE, Nov. 2021, pp. 581–592.

[19] X. Yin, Y. Feng, Q. Shi, Z. Liu, H. Liu, and B. Xu, "Fries: Fuzzing rust library interactions via efficient ecosystem-guided target generation," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. Vienna Austria: ACM, Sep. 2024, pp. 1137–1148.

[20] P. Liu, G. Zhao, and J. Huang, "Securing unsafe rust programs with xrust," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. Seoul South Korea: ACM, Jun. 2020, pp. 234–245.

[21] Z. Li, J. Wang, M. Sun, and J. C. Lui, "Mirchecker: Detecting bugs in rust programs via static analysis," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. Virtual Event Republic of Korea: ACM, Nov. 2021, pp. 2183–2196.

[22] Y. Bae, Y. Kim, A. Askar, J. Lim, and T. Kim, "Rudra: Finding memory safety bugs in rust at the ecosystem scale," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. Virtual Event Germany: ACM, Oct. 2021, pp. 84–99.

[23] B. Qin, Y. Chen, H. Liu, H. Zhang, Q. Wen, L. Song, and Y. Zhang, "Understanding and detecting real-world safety issues in rust," *IEEE Transactions on Software Engineering*, vol. 50, no. 6, pp. 1306–1324, Jun. 2024.

[24] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 190–200, Jun. 2005.

[25] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for javascript," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. Saint Petersburg Russia: ACM, Aug. 2013, pp. 488–498.

[26] D. Lehmann and M. Pradel, "Wasabi: A framework for dynamically analyzing webassembly," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. Providence RI USA: ACM, Apr. 2019, pp. 1045–1058.

[27] A. Eghbali and M. Pradel, "Dynapyt: A dynamic analysis framework for python," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Singapore Singapore: ACM, Nov. 2022, pp. 760–771.

[28] "Xray instrumentation — llvm 21.0.0git documentation," https://llvm.org/docs/XRay.html.

[29] "Instrumentation-based code coverage - the rustc book," https://doc.rust-lang.org/rustc/instrument-coverage.html.

[30] "Tracing - rust," https://docs.rs/tracing/latest/tracing/index.html.

[31] "What is ownership? - the rust programming language," https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html.

[32] "Generic types, traits, and lifetimes - the rust programming language," https://doc.rust-lang.org/book/ch10-00-generics.html.

[33] "The hir (high-level ir) - rust compiler development guide," https://rustc-dev-guide.rust-lang.org/hir.html.

[34] "The mir (mid-level ir) - rust compiler development guide," https://rustc-dev-guide.rust-lang.org/mir/index.html.

[35] "Non-lexical lifetimes (nll) fully stable | rust blog," https://blog.rust-lang.org/2022/08/05/nll-by-default/.

[36] T. Ball, "The concept of dynamic analysis," *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 6, pp. 216–234, Nov. 1999.

[37] J. Whaley, "Partial method compilation using dynamic profile information," *ACM SIGPLAN Notices*, vol. 36, no. 11, pp. 166–179, Nov. 2001.

[38] U. Bayer, A. Moser, C. Kruegel, and E. Kirda, "Dynamic analysis of malicious code," *Journal in Computer Virology*, vol. 2, no. 1, pp. 67–77, Aug. 2006.

[39] J. Hollingsworth, B. Miller, and J. Cargille, "Dynamic program instrumentation for scalable performance tools," in *Proceedings of IEEE Scalable High Performance Computing Conference*. Knoxville, TN, USA: IEEE Comput. Soc. Press, pp. 841–850.

[40] *A Generic and Configurable Source-Code Instrumentation Component*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 696–705.

[41] Z. Wang, A. Sanchez, and A. Herkersdorf, "Scisim: A software performance estimation framework using source code instrumentation," in *Proceedings of the 7th International Workshop on Software and Performance*. Princeton NJ USA: ACM, Jun. 2008, pp. 33–42.

[42] A. Bouchhima, P. Gerin, and F. Petrot, "Automatic instrumentation of embedded software for high level hardware/software co-simulation," in *2009 Asia and South Pacific Design Automation Conference*. Yokohama, Japan: IEEE, Jan. 2009, pp. 546–551.

[43] J. Maebe, M. Ronsse, and K. D. Bosschere, "Diota: Dynamic instrumentation, optimization and transformation of applications."

[44] A. R. Bernat and B. P. Miller, "Anywhere, any-time binary instrumentation," in *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*. Szeged Hungary: ACM, Sep. 2011, pp. 9–16.

[45] Z. Deng, X. Zhang, and D. Xu, "Spider: Stealthy binary program instrumentation and debugging via hardware virtualization," in *Proceedings of the 29th Annual Computer Security Applications Conference*. New Orleans Louisiana USA: ACM, Dec. 2013, pp. 289–298.

[46] "Unsafe rust - the rust programming language," https://doc.rust-lang.org/book/ch20-01-unsafe-rust.html.

[47] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker."

[48] A. Srivastava and A. Eustace, "Atom: A system for building customized program analysis tools," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*. Orlando Florida USA: ACM, Jun. 1994, pp. 196–205.

[49] W. Dietz, P. Li, J. Regehr, and V. Adve, "Understanding integer overflow in c/c++," *ACM Transactions on Software Engineering and Methodology*, vol. 25, no. 1, pp. 1–29, Dec. 2015.

[50] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 263–272, Sep. 2005.

[51] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Chicago IL USA: ACM, Jun. 2005, pp. 213–223.

[52] K. Kim, I. L. Kim, C. H. Kim, Y. Kwon, Y. Zheng, X. Zhang, and D. Xu, "J-force: Forced execution on javascript," in *Proceedings of the 26th International Conference on World Wide Web*. Perth Australia: International World Wide Web Conferences Steering Committee, Apr. 2017, pp. 897–906.

[53] F. Peng, Z. Deng, X. Zhang, and D. Xu, "X-force: Force-executing binary programs for security applications."

[54] "Auto merge of #117372 - amanieu:stdarch_update, r=mark-simulacrum · rust-lang/rust@ea37e80," https://github.com/rust-lang/rust/commit/ea37e8091fe87ae0a7e204c034e7d55061e56790.

[55] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," *ACM SIGPLAN Notices*, vol. 17, no. 6, pp. 120–126, Jun. 1982.

[56] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999.

[57] V. Nitin, A. Mulhern, S. Arora, and B. Ray, "Yuga: Automatically detecting lifetime annotation bugs in the rust language," *IEEE Transactions on Software Engineering*, vol. 50, no. 10, pp. 2602–2613, Oct. 2024.

[58] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi, "Disl: A domain-specific language for bytecode instrumentation," in *Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development*. Potsdam Germany: ACM, Mar. 2012, pp. 239–250.

[59] C. Flanagan and S. N. Freund, "The roadrunner dynamic analysis framework for concurrent programs," in *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. Toronto Ontario Canada: ACM, May 2010, pp. 1–8.

[60] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* San Francisco, CA, USA: IEEE Comput. Soc.

[61] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 89–100, Jun. 2007.