

Fuzzing for Rust’s Automatic Differentiation Crates

Austin Guiney
Computer Science & Engineering
University of California, San Diego
San Diego, USA
aguiney@ucsd.edu

Shaurya Raswan
Computer Science & Engineering
University of California, San Diego
San Diego, USA
sraswan@ucsd.edu

Matthew Beaubien
Computer Science & Engineering
University of California, San Diego
San Diego, USA
mbeaubien@ucsd.edu

Abstract—As Rust gains traction in high-performance computing (HPC), the reliability of its emerging automatic differentiation (AD) crates remains an important concern. In this work, we present `fuzz_core`, a differential fuzzing framework designed to identify numerical bugs in Rust’s AD libraries. By using PyTorch as a reference oracle and implementing a hybrid error tolerance model, we identified implementation bugs in two AD crates: `ad_trait` and `evalexpr_jit`. Our analysis uncovered several bugs across both libraries, including incorrect optimizations of derivatives, undefined behavior (UB) in trigonometric functions, and mathematical incorrectness in operator overloading. Our findings highlight the necessity of using differential testing to improve Rust’s readiness for use in machine learning (ML) applications.

Index Terms—Rust, Fuzzing, Automatic Differentiation, Software Testing, Safety, Machine Learning, Deep Learning, Undefined Behavior

I. INTRODUCTION

The machine learning (ML) community currently uses Python to develop inference systems for real-time applications because of its extensive library support for numerical analysis and automatic differentiation (AD), which include popular deep learning (DL) libraries such as PyTorch and TensorFlow [1]–[3]. Despite strong institutional support by ML developers, Python has limitations in high-performance computing (HPC) applications, ranging from slow execution, performance overhead from memory fragmentation, and restricted support for concurrency due to its Global Interpreter Lock (GIL) [1], [2].

Rust is an ideal language for HPC applications because it prevents memory leaks with its ownership model, offers memory-safe concurrency with its borrow checker, and prevents unsafe memory access by tracking the lifetime of references [1]. Rust clearly has advantages over Python for developing inference systems but the ML community has been slow to embrace Rust in HPC because it does not have a well-supported AD library [4].

The `ad_trait` and `evalexpr_jit` crates are emerging AD libraries for Rust that offer forward mode and reverse mode AD optimizations for HPC applications but they have not been well-tested [4], [5]. Previous research has used differential fuzzing to identify numerical bugs in several mature DL libraries, including PyTorch and TensorFlow [6].

We aim to identify bugs in `ad_trait` and `evalexpr_jit` using differential fuzzing in order to improve AD support in Rust and encourage more ML developers to use Rust in HPC

applications. We introduce the `fuzz_core` crate in order to support our effort in identifying numerical bugs in `ad_trait` and `evalexpr_jit` [7].

II. BACKGROUND

A. Automatic Differentiation

AD is a technique for computing the derivative of a function in a program by decomposing the function into a composition of elementary operations with known derivatives [3]. Repeated applications of the chain rule on the gradient of the decomposed function can then be used to calculate the derivative of the composition of elementary operations [3].

AD has been used by ML developers to calculate the derivatives of complex algorithms with control flow statements using constant computational complexity, which provides efficiency for HPC applications over other expensive numeric and symbolic differentiation methods [3]. Improving the quality of existing AD libraries in Rust would enable ML developers to take advantage of AD while benefiting from Rust’s fast execution speed, memory-safe concurrency, and ownership model that eliminates latency spikes in inference systems [1].

B. Forward Mode vs Reverse Mode

AD libraries have two methods of computing the derivatives of functions: forward mode and reverse mode [6].

Forward mode AD computes the derivative of the entire function by evaluating the chain rule on each elementary operation from the decomposed function from input to output, with the limitation that it can only compute the derivative of one input on one forward pass [3], [6]. Forward mode AD is therefore efficient for functions with few inputs and many outputs ($f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ where $n \ll m$) [3], [6].

Reverse mode AD computes the derivative of the entire function in the opposite way as forward mode AD, evaluating the chain rule on each elementary operation from the decomposed function from output to input [3]. Reverse mode AD computes the derivative of the entire function in two phases: a forward phase and a backward phase [6]. The forward phase builds a dependency graph from function values and the backward phase calculates the derivative of a function from output to input [3], [6].

Reverse mode AD has a similar limitation as forward mode AD because it may need multiple reverse passes to compute the derivative of a function [6]. Reverse mode AD is therefore

efficient for functions with many inputs and few outputs ($f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ where $n \gg m$) [3], [6].

Since forward mode AD and reverse mode AD both represent different optimizations for computing derivatives of functions using different accumulation orders of the chain rule on a decomposed function, they should produce the same result when used on the same function because matrix multiplication is an associative operation [3], [6].

C. Differential Fuzzing

Fuzzing is a popular technique for detecting security vulnerabilities in software by feeding generated inputs to a program under test (PUT) that may be malformed, either semantically or syntactically [8]. A reference oracle is a program that determines whether a PUT has a bug or security violation [8].

Fuzzing is not sufficient for detecting implementation bugs that fail to crash a program because researchers have to manually check whether or not the PUT produces a correct output [9].

Differential testing is a technique that exposes bugs in programs by comparing two programs that implement the same specification and testing whether both programs produce the same output for the same input [10]. We can use differential testing with fuzzing by feeding the same malformed input to two different PUTs that implement the same specification, which we will call differential fuzzing [6], [10]. We know an implementation bug exists when both PUTs return different results for the same input because each PUT acts as a reference oracle for the other PUT [6].

We can use fuzzing in Rust by using the cargo-fuzz crate in a program and compiling it against Rust’s nightly build [11]. cargo-fuzz is a program that invokes the libfuzzer-sys crate, a wrapper for LLVM’s fuzzing library for C++ [11], [12].

D. Existing Approaches to AD in Rust

Existing approaches to implementing AD in Rust include the tch-rs crate, which implements a Rust binding for LibTorch [13]. LibTorch itself implements a C++ binding for the PyTorch library, a mature DL library that supports AD [1], [13].

The primary limitation of tch-rs is that it breaks Rust’s safety guarantees because it relies on a foreign function interface (FFI) to a C++ library, which can potentially introduce undefined behavior (UB) in a Rust program that uses the crate [14]. Relying on the tch-rs crate for AD therefore eliminates the advantages of using Rust for HPC applications and inference systems, which means that a native AD library for Rust is needed to take advantage of its memory-safe guarantees [14].

E. *ad_trait*

The *ad_trait* crate introduces AD in native Rust by implementing a trait that can accumulate the necessary information for computing the derivative of a function [4]. *ad_trait* supports both forward mode and reverse mode AD, which means that we can use differential fuzzing on the crate to identify

implementation bugs in either optimization [4]. The forward and reverse mode ADs for *ad_trait* serve as two of our PUTs for *fuzz_core*.

F. *evalexpr_jit*

The *evalexpr_jit* crate is a mathematical expression evaluator that uses just-in-time compilation (JIT) and AD to compute partial derivatives and Jacobian matrices [5]. *evalexpr_jit* serves as another PUT for *fuzz_core*.

G. PyTorch as a Reference Oracle

PyTorch is a mature DL library in Python that supports dynamic tensor compilations with support for AD and hardware acceleration while matching the performance of other fast DL libraries [15].

PyTorch stands apart from other DL libraries by using define-by-run execution, which differentiates functions by directly running the computation as opposed to creating a static graph ahead of time [16]. PyTorch’s define-by-run-execution gives developers the ability to use arbitrary control-flow constructs in exchange for additional overhead because it needs to calculate derivatives during each pass of the AD compilation instead of using a static graph [16].

PyTorch’s maturity and performance advantages over other static DL libraries makes it an ideal use as a reference oracle for our differential fuzzing of *ad_trait* and *evalexpr_jit* [1], [15], [16]. Prior research by Yang also used PyTorch as a reference oracle for differentially fuzzing DL libraries with AD support, validating our decision to use it for *fuzz_core* [6].

For the purposes of *fuzz_core*, we will assume that PyTorch gives the accepted value for an AD compilation so that we can determine which optimization modes in *ad_trait* have implementation bugs. Although prior research by Yang found PyTorch itself to have implementation bugs in its AD calculations, PyTorch’s maturity and widespread use by the ML community makes it a trusted source for establishing a reference oracle for our project [1], [6].

III. METHODOLOGY

A. The *fuzz_core* crate

We created a Rust crate called *fuzz_core* in order to differentially fuzz the *ad_trait* and *evalexpr_jit* crates. We organized our project into a crate so that we can differentially fuzz multiple AD crates in Rust with minimal code reuse.

We can differentially fuzz a new AD library by creating an entry-point program called a fuzz target that uses the *fuzz_core* crate. Our *fuzz_core* crate implements a simple pipeline:

- 1) The cargo-fuzz crate generates a stream of random bytes.
- 2) Our input decoder converts the random bytes from cargo-fuzz into N 64-bit (f64) floating-point (FP) inputs.
- 3) We create an abstract syntax tree (AST) to generate a random mathematical function using the N f64 FP inputs from our input decoder as arguments.
- 4) Our fuzzing harness computes the derivative of our random function using PyTorch.

- 5) Our reference oracle compare the output of our PUT to the output of our reference oracle.

B. Input Generation

Our input decoder generates inputs by interpreting a prefix of the random byte stream from cargo-fuzz as a sequence of floating-point values.

Given N variables, the decoder consumes the initial $N \times 8$ bytes of the stream. It splits the sequence into 8-byte chunks and interprets each chunk as an IEEE 754 double-precision (f64) value. We construct an input vector $\mathbf{x} \in \mathbb{R}^N$ to evaluate the generated AST.

C. AST Generation

We use the arbitrary crate to automatically generate random functions. The core data structure that we use is the `Expr` enum, which represents a recursive AST composed of unary and binary operations that terminates with variables or constants. The definition of `Expr` enum can be seen in Listing 1.

```
1 pub enum Expr<T> {
2     Number(T, f64),
3     Id(T, String),
4     UnOp(T, Op1, Box<Expr<T>>),
5     BinOp(T, Op2, Box<Expr<T>>, Box<Expr<T>>),
6 }
```

Listing 1. Expr enum implementation

The generation process consumes cargo-fuzz’s byte stream as an `Unstructured` source to make deterministic, probabilistic choices. We constrain this process using an `AstGenConfig` struct, which limits tree depth and operator selection. The generation algorithm follows three key phases:

- 1) **Recursive Structure:** The generator builds nested `Expr` nodes until it reaches the maximum depth or probabilistically selects a terminal.
- 2) **Variable Scoping:** We maintain a stack of active variables to ensure that the expression is well-formed. When generating a terminal, the algorithm probabilistically decides whether to reuse an existing variable from the stack or introduce a new one. This ensures that variables are introduced sequentially and can be reused with a realistic distribution.
- 3) **Operator Selection:** We implement the `Arbitrary` trait for operation enums (`Op1`, `Op2`), which allows the fuzzing harness to use the input bytes on a range of mathematical functions.

The final output is a `GeneratedExpr` struct, which contains the root of the AST, the encapsulated variable environment, and the input arity. Our structure gives our fuzzing harness the necessary metadata to execute the expression across different AD libraries.

D. Fuzzing Harness

We created a fuzzing harness to differentially test `ad_trait` and `evalexpr_jit` using Rust’s trait system. A fuzzing harness is a program that acts as an adapter between the fuzzing engine and the PUT [17]. The fuzzing harness is responsible for initializing the testing environment and translating the fuzzer’s raw input bytes into function calls to the PUT [17].

We define a polymorphic interface, `MainBackend`, which specifies the abstract operations required for differentiation, including trigonometric functions, basic arithmetic, and zero initialization. In Listing 2, the `MainBackend` trait enforces a contract that every AD library must use.

```
1 pub trait MainBackend: Sized + Clone {
2     fn from_f64(val: f64) -> Self;
3     fn zero() -> Self;
4     fn cos(self) -> Self;
5     ...
6 }
7
8 pub fn evaluate<T: MainBackend, Tag>(
9     expr: &Expr<Tag>,
10    env: &Env<T>,
11 ) -> Result<T, String> {
12     match expr {
13         Expr::Number(_, val) => Ok(T::
14             from_f64(*val)),
15         ...
16     }
17 }
```

Listing 2. MainBackend Trait Implementation

The `MainBackend` trait serves as the interface for all evaluation engines.

By enforcing a standard contract of operations for every target library, we decouple the definition of a mathematical expression from its execution. Our system allows the fuzzing harness to traverse a single generated AST and dispatch operations to multiple backends at the same time, which include traits such as `Calculator` and `PyTorchComputable`.

Since our system can evaluate the same input against multiple AD libraries, we can ensure that any differences in the outputs are due to implementation bugs rather than different inputs.

E. Defining Oracles

We defined two types of oracles for `fuzz_core`: a reference oracle and a metamorphic oracle [8], [9].

Our metamorphic oracle compares both of `ad_trait`’s optimization modes against each other [9]. By treating each AD mode optimization as the oracle for the other optimization, we can identify implementation bugs in `ad_trait` without needing to know the accepted value for the derivative of our random function [3], [6].

Our reference oracle uses `PyTorch` as the accepted value for the derivative of our random function because it is well-tested and has performance advantages over other DL libraries

[15], [16]. We individually compared PyTorch’s computation to each optimization mode in `ad_trait` and `evalexpr_jit`’s AD computation to identify bugs in each library. Our reference oracle helped us determine the root cause for the implementation bugs in `ad_trait` after discovering them with metamorphic testing.

F. Defining Hybrid Error Tolerances

Our oracles need to set an error tolerance threshold in order to determine whether our PUTs have implementation bugs [18]. Our oracles cannot use the `assert_eq!` macro because PyTorch accumulates gradients in a non-deterministic order and FP arithmetic is non-associative by definition [16], [19]. We therefore need a hybrid error tolerance model that determines a threshold for how much our oracle and PUT can differ before we can determine whether or not our PUT has an implementation bug [18].

We defined the following error tolerance model for our oracles [18]:

$$|y_{ref} - y| \leq \epsilon_{abs} + \epsilon_{rel} \cdot |y_{ref}|$$

We set ϵ_{abs} to 10^{-12} , which is the absolute error tolerance for an oracle. The absolute error tolerance is used when FP numbers are close to 0 because relative errors grow unbounded when FP numbers approach 0 [18].

We set ϵ_{rel} to 10^{-9} , which is the relative error tolerance for an oracle. The relative error tolerance is used when both FP numbers are large because absolute errors grow unbounded when both FP numbers approach infinity [18].

We defined our tolerances based on the theoretical limits of IEEE 754 double-precision arithmetic, selecting conservative bounds that allow for accumulation error while remaining stricter than standard ML libraries [16], [18], [20].

G. Fuzz targets

We created three fuzz targets that use `fuzz_core` to differentially fuzz `ad_trait` and `evalexpr_jit`.

- 1) **fuzz_target_1**: An initial prototype using reverse polish notation (RPN) to generate random functions. We deprecated this target in favor of using random AST generation to support more complex equations.
- 2) **fuzz_target_ast**: The primary target that differentially fuzzes `ad_trait` using ASTs to generate random functions.
- 3) **fuzz_target_evalexpr_jit**: A secondary target that differentially fuzzes `evalexpr_jit` using ASTs to generate random functions.

Our fuzz targets support the following command line arguments during runtime:

- 1) **FUZZ_MODE**: Controls the failure handling strategy. The default `PanicOnFirstError` calls the `panic!` macro to halt execution when our reference oracle discovers an implementation bug in an AD library. We can alternatively use `continuous` to ignore failures so that the user can find multiple implementation bugs.

- 2) **FUZZ_TARGET**: Selects the reference oracle:

- `rev_fwd`: Differentially fuzzes `ad_trait` using a metamorphic oracle.
- `rev_gt / fwd_gt`: Differentially fuzzes one of `ad_trait`’s optimization modes against our reference oracle.
- `all` (Default): Differentially fuzzes `ad_trait` using both metamorphic and reference oracles.

- 3) **FUZZ_TESTS**: Specifies the batch size of random functions generated per input stream from cargo-fuzz (Default: 1).

```

1  #![no_main]
2  use libfuzzer_sys::fuzz_target;
3
4  fuzz_target!(|data: &[u8]| {
5      if let Ok(s) = std::str::from_utf8(data)
6      {
7          // Attempt to differentiate
8          let _ = my_ad_crate::diff(s);
9      }
10 });

```

Listing 3. Example Fuzz Target

IV. EVALUATION

A. Results

1) *General Observations*: The following examples highlight specific issues that we encountered when differentially fuzzing `ad_trait`:

$$f(x) = \frac{x}{x} \quad (1)$$

$$f(x) = \cos\left(\frac{x}{x}\right) \quad (2)$$

- **FP Error Accumulation**: Forward and reverse passes exhibit differing FP error accumulation due to how they traverse the computational graph.

- **Undefined Behavior**: We discovered UB when evaluating $\cos(\frac{x}{x})$.

2) *Forward Mode Edge Cases*: We also identified several issues within `ad_trait`’s `ForwardAD` mode:

- **Incorrect derivatives** (`acosh`): The implementation of the derivative for `acosh` incorrectly decomposes $\frac{1}{\sqrt{x^2-1}}$ as $\frac{1}{\sqrt{x-1}} \cdot \frac{1}{\sqrt{x+1}}$.
- **Logarithm Function Handling**: The derivative of the natural logarithm is defined as:

$$\frac{d}{dx} \ln(|x|) = \frac{1}{x}$$

The derivative of $\ln$ is well-defined for $x \neq 0$ and should be able to accept negative values of x as inputs. However, the derivative for the `log` function in `ForwardAD` mode does not handle negative inputs and assumes that $x > 0$, resulting in NaN. We fixed this by accounting for the edge case where the `log` function has an undefined derivative.

- **Divergent Branching in Square Root Functions:** The `sqrt` function exhibits divergent branching with UB for negative inputs. The derivative of $\frac{d}{dx}(\sqrt{x})$ correctly results in NaN for $x < 0$. However, when the derivative of this function evaluates to 0, `ForwardAD` still calculates the derivative as NaN and propagates the result during the forward pass. This behavior incorrectly resulted in derivatives being evaluated as NaN.

For example, the derivative:

$$\frac{d}{dx}(\sqrt{-1} + x) = 1$$

should not result in NaN despite containing a complex number in the original expression.

3) *evalexpr_jit*:

- **AST fix:** We identified a bug in `evalexpr_jit`'s implementation of `Expr::Sqrt`, which incorrectly calculated derivatives with the form:

$$\frac{d}{dx}(\sqrt{x}) = \frac{1}{2\sqrt{x}}$$

The AST for the derivative of `Expr::Sqrt(expr)` did not include the constant multiplication $\frac{1}{2}$, leading to an incorrect AST construction before JIT optimization passes.

We have submitted bug reports to the maintainers of `evalexpr_jit` and `ad_trait` for each issue that we found.

B. Case Study 1: Competing fract implementations in `ad_trait`

Our metamorphic testing of the `ad_trait` crate uncovered an implementation bug in the forward mode AD optimization for the `fract` function.

The `ad_trait` crate does not use a single source of truth (SSOT) between the forward AD and reverse mode AD optimizations. Rather than organizing code that is common to both optimizations into a single crate, `ad_trait` re-implements the same mathematical functions several times across files: `adfn.rs`, `adf.rs`, `adr.rs`, and `adr2.rs`. Each of these files re-implement the same function twice for each AD mode optimization.

```
1 #[inline]
2 fn fract(self) -> Self {
3     Self::new(self.value.fract(), [1.0; N])
4 }
```

Listing 4. Incorrect fract implementation for forward mode AD in `ad_trait`.

Listing 4 shows an incorrect implementation of the `fract` function in `adfn.rs`, a math crate for the forward mode AD optimization in `ad_trait`. The `fract` function is defined as the fractional part of a FP number [21]:

$$\text{fract}(x) = x - \lfloor x \rfloor$$

The code in Listing 4 should return the derivative of the `fract` function, which is defined as [21]:

$$\frac{d}{dx}(\text{frac}(x)) = 1 \quad \text{for } x \notin \mathbb{Z}$$

In `adfn.rs`, the derivative of `fract` is hardcoded to be 1, which ignores the chain rule when forward mode AD calculates the derivative of a composite functions involving fractions. We need to use the chain rule in order to correctly calculate the derivative of the `fract` function composed with another function [21]:

$$y = \text{fract}(u(x))$$

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx} = 1 \cdot \frac{du}{dx}$$

Listing 5 shows the correct implementation for the derivative of the `fract` function for the reverse mode AD optimization in `ad_trait`:

```
1 NodeType::Fract => {
2     tiny_vec!([f64; 2]) => 1.0)
3 }
```

Listing 5. Correct fract implementation for reverse mode AD in `ad_trait`.

We see that Listing 5 correctly implements the derivative of the `fract` function by calculating the local gradient of the `fract` function and passing the calculation to other nodes.

C. Case Study 2: Incorrect derivative of `acosh` function in `ad_trait`

When examining the source code for the reverse mode AD for `ad_trait`, we discovered an implementation bug for the `acosh` function in `adr.rs`.

The `acosh` function is defined as the inverse hyperbolic cosine function [21]. The derivative of the `acosh` function is defined as [21]:

$$\frac{d}{dx}(\text{acosh}(x)) = \frac{1}{\sqrt{x^2 - 1}}$$

Listing 6 shows the implementation of the derivative of the `acosh` function in `adr.rs`:

```
1 NodeType::Acosh => {
2     let lhs = parent_0.unwrap();
3     tiny_vec!([f64; 2]) => 1.0/(ComplexField
4         ::sqrt(lhs - 1.0)*ComplexField::
5         sqrt(lhs + 1.0)) )
6 }
```

Listing 6. Incorrect derivative of the `acosh` function in `ad_trait`.

In Listing 6, `adr.rs` calculates the derivative of the `acosh` function by decomposing the function into a product of $\frac{1}{\sqrt{x-1}}$ and $\frac{1}{\sqrt{x+1}}$.

The implementation of the derivative of `acosh` function in Listing 6 is mathematically equivalent to the standard implementation of the derivative when $x > 1$ but leads to a sign error when $x < 1$ [22]. For example, if $x = -2$, then derivative of the `acosh` function should be:

$$\frac{d}{dx}(\text{acosh}(-2)) = \frac{1}{\sqrt{(-2)^2 - 1}} = \frac{1}{\sqrt{4 - 1}} = \frac{1}{\sqrt{3}}$$

ad_trait evaluates the derivative of acosh function incorrectly by flipping the sign of the result when $x < 0$ due to an implementation bug:

$$\frac{d}{dx}(\text{acosh}(-2)) = \frac{1}{\sqrt{-2-1}} \cdot \frac{1}{\sqrt{-2+1}} = -\frac{1}{\sqrt{3}}$$

There is a lack of consistency in the way that the developers compute terms with a square root. For example, the asinh function is defined as the inverse hyperbolic sin function. The derivative of the asinh function is defined as [21]:

$$\frac{d}{dx}(\text{asinh}(x)) = \frac{1}{\sqrt{x^2 + 1}}$$

In Listing 7, the reverse mode AD implements the derivative of the asinh function without decomposing the $\sqrt{x^2 + 1}$ term:

```
1 NodeType::Asinh => {
2   let lhs = parent_0.unwrap();
3   tiny_vec!([f64; 2]) => 1.0/(lhs*lhs +
4     1.0).sqrt())
5 }
```

Listing 7. Standard implementation of the derivative of the asinh function in ad_trait.

The inconsistency in the way that the ad_trait crate computes functions with square roots indicates that the crate suffers from manual optimizations of mathematical functions. We have highlighted some of the benefits of differential fuzzing in this example.

V. CONCLUSION

We presented fuzz_core in this article, a differential fuzzing framework intended to support our effort in identifying numerical bugs in Rust’s AD libraries. By using PyTorch as a reference oracle and implementing a hybrid error tolerance model, we successfully identified several bugs in ad_trait, including incorrect implementations for the derivatives of fract and acosh. Our findings demonstrate that even languages with strong memory guarantees require metamorphic testing to reduce mathematical errors in HPC applications.

Our results highlight that AD libraries in Rust are not yet mature enough to challenge established DL libraries in Python, including PyTorch. Differential fuzzing is therefore a necessary tool to ensure that Rust’s AD libraries are well-tested and implemented correctly.

A. Future Work

We identify three key avenues for extending this research:

- **API-Level Fuzzing:** While our AST generation targets the internal logic of mathematical operators, future iterations could adopt the byte-driven, property-based approach used by tools such as NablaFuzz [6]. Property-based approaches would allow us to explore edge cases at the API boundary, testing how libraries handle tensor shapes, broadcasting rules, and invalid memory layouts.

- **Optimization Oracles:** Currently, fuzz_core detects correctness bugs. We propose extending the metamorphic oracle to detect missed optimizations by comparing the execution traces of JIT-compiled graphs against hand-optimized kernels. Our work is especially relevant for libraries such as evalexpr_jit that rely on runtime optimization passes [5].
- **Compiler Integration:** As Rust’s experimental autodiff feature matures, we aim to extend our backend to target Enzyme [23], [24]. Since Enzyme operates as an LLVM pass, applying differential fuzzing at this level would allow us to verify the correctness of gradients generated directly by the compiler backend and beyond library-level verification [24].

REFERENCES

- [1] R. Mishra, “High-performance machine learning inference systems with rust: A review of techniques, tools, and optimizations,” *International Research Journal of Engineering and Technology (IRJET)*, vol. 11, no. 11, pp. 639–644, Nov 2024.
- [2] S. Singh, “Comparative Analysis of Python’s Role in AI and Machine Learning,” *International Journal of Research in Engineering, Science and Management*, vol. 8, no. 2, pp. 59–64, Feb 2025. [Online]. Available: <https://journal.ijresm.com/index.php/ijresm/article/view/3219>
- [3] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, “Automatic differentiation in machine learning: a survey,” *arXiv preprint arXiv:1502.05767*, 2015. [Online]. Available: <https://arxiv.org/abs/1502.05767>
- [4] C. Liang, Q. Wang, A. Xu, and D. Rakita, “ad-trait: A fast and flexible automatic differentiation library in rust,” Apr 2025.
- [5] J. Range, “Jr-1991/evalexpr-jit,” <https://github.com/JR-1991/evalexpr-jit>, Jul 2025. [Online]. Available: <https://github.com/JR-1991/evalexpr-jit>
- [6] C. Yang, Y. Tu, Y. Deng, H. Li, J. Yao, and L. Zhang, “Fuzzing automatic differentiation in deep-learning libraries,” Feb 2023.
- [7] A. Guiney, S. Rawson, and M. Beaubien, “pyreking/cse291y,” <https://github.com/pyreking/cse291y>, Dec 2025. [Online]. Available: <https://github.com/pyreking/cse291y>
- [8] V. J. M. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, “The art, science, and engineering of fuzzing: A survey,” 2019. [Online]. Available: <https://arxiv.org/abs/1812.00140>
- [9] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, “The oracle problem in software testing: A survey,” *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015.
- [10] W. M. McKeeman, “Differential testing for software,” *Digital Technical Journal*, vol. 10, pp. 100–107, 1998. [Online]. Available: <https://api.semanticscholar.org/CorpusID:14018070>
- [11] N. Fitzgerald, M. Goregaokar, C. Farwell *et al.*, “rust-fuzz/cargo-fuzz,” <https://github.com/rust-fuzz/cargo-fuzz>, Nov 2025, gitHub repository. [Online]. Available: <https://github.com/rust-fuzz/cargo-fuzz>
- [12] N. Fitzgerald, C. Farwell, M. Goregaokar, S. Kazlauskas *et al.*, “rust-fuzz/libfuzzer,” <https://github.com/rust-fuzz/libfuzzer>, Jul 2025, gitHub repository. [Online]. Available: <https://github.com/rust-fuzz/libfuzzer>
- [13] L. Mazare, “Laurentmazare/tch-rs,” <https://github.com/LaurentMazare/tch-rs>, Nov 2025. [Online]. Available: <https://github.com/LaurentMazare/tch-rs>
- [14] The Rust Project Developers, *The Rust Reference*, The Rust Foundation, 2025, accessed: 2025-12-06. [Online]. Available: <https://doc.rust-lang.org/reference/>
- [15] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “Pytorch: An imperative style, high-performance deep learning library,” 2019. [Online]. Available: <https://arxiv.org/abs/1912.01703>
- [16] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, “Automatic differentiation in pytorch,” in *NIPS 2017 Autodiff Workshop*, 2017. [Online]. Available: <https://openreview.net/forum?id=BJJsrnFCZ>

- [17] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, “Fudge: Fuzz driver generation at scale,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019, pp. 975–985.
- [18] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. Del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, “Array programming with NumPy,” *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020.
- [19] D. Goldberg, “What every computer scientist should know about floating-point arithmetic,” *ACM Comput. Surv.*, vol. 23, no. 1, pp. 5–48, Mar 1991. [Online]. Available: <https://dl.acm.org/doi/10.1145/103162.103163>
- [20] N. J. Higham, 2. *Floating Point Arithmetic*. Philadelphia, PA: SIAM, 2002, pp. 35–60. [Online]. Available: <https://epubs.siam.org/doi/abs/10.1137/1.9780898718027.ch2>
- [21] “NIST Digital Library of Mathematical Functions,” <https://dlmf.nist.gov/>, Release 1.2.4 of 2025-03-15, f. W. J. Olver, A. B. Olde Daalhuis, D. W. Lozier, B. I. Schneider, R. F. Boisvert, C. W. Clark, B. R. Miller, B. V. Saunders, H. S. Cohl, and M. A. McClain, eds. [Online]. Available: <https://dlmf.nist.gov/>
- [22] J. W. Brown and R. V. Churchill, *Complex Variables and Applications*, 8th ed. Maidenhead, England: McGraw Hill Higher Education, Jan. 2008.
- [23] A. Leal, “autodiff,” 2018, accessed 2025-12-10. [Online]. Available: <https://github.com/autodiff/autodiff>
- [24] W. Moses and V. Churavy, “Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients,” in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 12472–12485. [Online]. Available: <https://proceedings.neurips.cc/paper/2020/file/9332c513ef44b682e9347822c2e457ac-Paper.pdf>