RustBuildEq: A Benchmark for Training and Evaluating Binary Equivalence Classifiers under Build Variability

Elliott Wen, Chenye Ni, Valerio Terragni
elliott.wen,chenye.ni,v.terragni@auckland.ac.nz
University or Auckland
New Zealand

Jens Dietrich
jens.dietrich@vuw.ac.nz
Victoria University of Wellington
New Zealand

Abstract

Reproducible independent rebuilds strengthen software supply-chain integrity by recreating the original build environment and enforcing bitwise equivalence between artifacts. However, this approach implicitly assumes a trustworthy toolchain and can fail under adversarial manipulation of the build process itself (e.g., the Ken Thompson attack). Prior work has explored introducing diversity across build environments to reduce reliance on any single toolchain, and has proposed AI-driven methods to establish behavioural equivalence while tolerating benign build variability in the Java ecosystem. In this work, we extend this line of research to Rust and present *RustBuildEq*, a benchmark for training and evaluating binary equivalence classier models under realistic build variability. We curate a large corpus of crates drawn from the top 20% of the crates.io ecosystem and construct datasets of equivalent (EQ) and non-equivalent (NEQ) pairs with rich provenance metadata. EQ pairs are generated from identical source revisions under varying toolchain versions and build configurations, while NEQ pairs are derived from AST rewrites or API-breaking changes across versions. Many rust crates rely heavily on generics and cannot be compiled into binaries without specifying concrete types; to address this, we develop an automated approach that combines heuristic type instantiation, witness-type synthesis, and an iterative AI repair loop. RustBuildEq comprises 19,184,671 EQ records and 273,848,531 NEQ records, and includes a Python API for dataset navigation. The dataset provides large-scale ground truth for training and evaluating AI-driven models for reasoning binary equivalence and is publicly available at https://doi.org/10.5281/zenodo.19244908[1].

ACM Reference Format:
Elliott Wen, Chenye Ni, Valerio Terragni and Jens Dietrich. 2026. RustBuildEq: A Benchmark for Training and Evaluating Binary Equivalence Classifiers under Build Variability. In . ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Reproducible builds [2, 7] are widely used to strengthen software supply-chain integrity by enabling independent parties to rebuild artifacts and verify their equality. Promoted by frameworks such as SLSA [5, 6], this approach aims to detect discrepancies indicative of tampering. However, it relies on replicating the original build environment and therefore implicitly assumes a trustworthy toolchain. In adversarial settings, this assumption may not hold: a compromised compiler or build tool can produce identical but still malicious artifacts, as illustrated by the Ken Thompson attack.

This limitation motivates a complementary perspective that relaxes strict equality in favour of controlled diversity in the build environment. By allowing variation across toolchains and configurations, reliance on any single trusted component can be reduced. However, this introduces build-induced variability, whereby artifacts derived from identical source code may differ at the binary level despite preserving intended semantics. Consequently, the problem shifts from verifying equality to determining equivalence. The central question becomes when two artifacts should be regarded as implementing the same program, and when observed differences indicate potential compromise. Recent work [4] has formalised binary equivalence as a machine-learning classification problem over pairs of artifacts, with the objective of determining whether they correspond to the same underlying program up to permissible build-induced variation. This line of work has also introduced benchmark datasets [3], notably for Java programs compiled to JVM bytecode, to support the training and evaluation of equivalence classifiers.

In this work, we extend the study of binary equivalence to Rust, a language that has seen widespread adoption in production environments and whose artifacts are frequently distributed as pre-built binaries across heterogeneous platforms. In such settings, independent rebuilds and third-party provenance verification are increasingly common, making robust artifact comparison practically important. At the same time, Rust build pipelines introduce sources of variability that are not present in JVM-based ecosystems. Rust binaries are produced through multi-stage compilation pipelines that incorporate backend code generation, whole-program linking, and optional post-link optimisation. Consequently, artifacts may differ due to factors such as toolchain evolution, optimisation settings, link-time transformations, and the inclusion or removal of debugging information, metadata, and custom sections.

We present a new benchmark for training and evaluating binary equivalence classifiers in Rust under realistic build conditions. Our corpus consists of widely used crates drawn from the top 20% of the ecosystem by popularity. From this corpus, we derive an equivalence dataset by compiling identical source revisions under diverse real-world configurations (e.g., toolchain versions, debugging information settings, and code generation backends), yielding approximately 19 million artifact pairs. We further construct a complementary non-equivalence dataset by pairing artifacts that differ due to semantics-changing AST rewrites or API-breaking changes

[1] Owing to its large size (over 67 TB), we welcome requests for full access and can assist in arranging physical hard drive transfer if needed.

Conference'17, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Category	Crate	Version	Download Count
Serialization	`serde`	`1.0.228`	`853,551,780`
Async runtime	`tokio`	`1.50.0`	`552,594,042`
CLI	`clap`	`4.5.60`	`699,338,199`
HTTP client	`reqwest`	`0.13.2`	`390,104,649`
Randomness	`rand`	`0.10.0`	`937,119,179`
Logging	`tracing`	`0.1.44`	`491,617,273`
Crypto	`ring`	`0.17.14`	`429,642,673`
Database	`diesel`	`2.3.6`	`22,801,703`

Table 1: Example crates and corresponding download counts from selected representative categories.

across versions, resulting in more than 270 million pairs. Many Rust libraries rely heavily on generics and cannot be compiled into binaries without specifying concrete types; to address this, we employ a combination of automated techniques, including heuristic type instantiation, witness-type synthesis, and a repair loop assisted by a large language model. Together, these datasets provide large-scale ground truth for training and evaluating models that reason about binary equivalence under realistic variability. They support the development of AI-driven systems that assist or directly perform equivalence reasoning within modern software supply-chain workflows. The benchmark is released alongside a lightweight Python API for querying metadata and retrieving labeled artifact pairs.

2 Benchmark Construction

As a first step, we curated a Rust crate corpus from a snapshot of the crates.io ecosystem taken on 15 March 2026. Crates.io is the canonical public registry and primary distribution channel for Rust crates, and thus provides representative coverage of the ecosystem for large-scale sampling. Using the crates.io API, we collected crate metadata, including category assignments and download counts. Crates.io defines a fixed taxonomy of categories [2]; at the time of collection, the ecosystem comprised 58 categories spanning domains such as networking, cryptography, and data processing, as illustrated in Table 1.

To ensure broad and representative coverage, we performed stratified sampling by category, selecting 20% of crates within each category with probabilities proportional to their download counts. Since crates may belong to multiple categories, the initial sample contained duplicates arising from overlapping category memberships. We therefore deduplicated the sample at the crate level to obtain a unique set of crates. For each sampled crate, we retrieved the published source by downloading the corresponding crates.io source tarball. After deduplication, the final curated corpus contains 31,026 unique crates.

2.1 Artifact Compilation

We next compile the curated crates into binary artifacts. Rust crates in our corpus fall into two main types: binary crates and library crates. This compilation process differs by crate type. Binary crates produce standalone executables directly. In contrast, library crates generate `rlib` files, which serve as intermediate artifacts for downstream linking. An `rlib` contains machine code for non-generic functions. However, for generic functions, it retains only Rust intermediate representations (i.e., MIR) along with metadata. These generic functions are lowered to concrete machine code only when downstream crates instantiate them with specific type arguments. This distinction is important for our corpus: according to our measurement, library crates account for 93% of all crates, and within them, generic functions comprise more than 57% of public APIs. As a result, compiling library crates into `rlib` files alone would leave a substantial portion of code in an unmaterialized form.

To address this issue, we generate a lightweight driver program for each library that exercises its public API as comprehensively as possible, thereby producing a fully linked binary. Specifically, we use Rust compiler internal type-system APIs (e.g., `TyCtxt::hir_crate_items`) to enumerate function signatures and retain only publicly visible functions. The driver then takes a reference to each public function by coercing it to a function pointer and printing its address, as shown in Listing 1 (line 19). For non-generic functions, this approach is sufficient to trigger machine code generation. Handling generic functions is more challenging, as they require appropriate concrete type arguments for instantiation (line 20). To address this, we employ three complementary strategies to instantiate these generics.

Heuristic Search: We leverage the compiler's type system to determine the number of generic parameters and their associated bounds for each generic function. We then construct a pool of candidate type combinations from primitive types and public types defined in the crate, and search this space to find an assignment that satisfies all trait and `where`-clause constraints. To control combinatorial blow-up, we first build a dependency graph over generic parameters. Each node is a generic parameter, and each edge represents a type-level dependency induced by bounds. For example, consider `fn f<T, U>() where T: Into<U>, U: Default`. Under this convention, the bound `T: Into<U>` adds `U -> T`, because `U` must be fixed before this bound can be checked. We then apply Kahn's topological sorting to order parameters from prerequisites to dependents. This means parameters with zero unmet dependencies are instantiated first, followed by parameters that depend on them. This ordering helps invalid branches fail early during candidate search. We also memoize failed prefixes (partial assignments of the first k parameters), so any later branch that shares the same prefix is pruned without re-running trait checks. This strategy is simple yet effective. In our preliminary observation, it can instantiate around 61% of public generic functions.

Witness Type Synthesis: For generic functions not resolved by heuristic search, we synthesize concrete witness types directly from their bounds. For each generic parameter, we collect trait bounds, supertraits, associated-type equalities, and where-clauses. We then generate a minimal `struct` or `enum` with the required trait implementations, associated-type definitions, and stub method signatures. For method bodies, we return default values when the return type is a known primitive or standard container. For more complex return types, we employ an unsafe fallback that allocates memory via a C `malloc` function through FFI and casts it to the target type. When bounds introduce nested obligations (e.g., `T: Iterator<Item = U>`), we recursively synthesize witness types for associated types in dependency order. We continue this process until the compiler trait solver accepts the full constraint set. Combined with heuristic

[2] https://crates.io/categories

Listing 1: Examplary driver program referencing public APIs from the target library.

```rust
trait DoThing {
    fn do_thing(&self) -> i32;
}

// Public APIs imported from the target library crate.
// foo:fn(i32)->i32  bar:fn<T:Copy>(T)->T baz:fn<T:DoThing>(T)->()
use target_crate::{foo, bar, baz};

// Synthesized witness type that satisfies trait bounds.
struct Synth;
impl DoThing for Synth {
    fn do_thing(&self) -> i32 { 0 }
}

fn main() {
    // Coerce to a non-generic function pointer.
    println!("{:p}", foo as *const ());
    // Generic function instantiated via heuristic search.
    println!("{:p}", bar::<u32> as *const ());
    // Generic function instantiated with a synthesized type.
    println!("{:p}", baz::<Synth> as *const ());
}
```

search, this strategy pushes the instantiation success rate to around 87%.

Large Language Model-Assisted Repair: Witness type synthesis still fails for a subset of complex generics. Two common cases are higher-ranked trait bounds (HRTB) and generic associated types (GAT). HRTB introduces universal lifetime requirements (e.g., `for<'a>`), which means one implementation must type-check for every lifetime, not just one guessed lifetime. Template-style local rewrites often pick a concrete lifetime shape, so they can pass a local check but still fail the required "for all lifetimes" condition. GAT introduces associated type families whose concrete forms depend on both trait implementations and generic parameters. These constraints are often coupled across multiple items, so a local synthesis step may miss the globally consistent assignment needed by the trait solver. For unresolved cases, we use an LLM-assisted repair loop. We use `Qwen3-Coder-480B-A35B-Instruct-FP8` as the repair model. The prompt includes four parts: the failing code snippet, the relevant function signature and bounds, the compiler error messages, and the instruction to return a minimal compilable patch. In each turn, we compile the generated driver, collect compiler diagnostics, and ask the model for a minimal patch that fixes the reported type errors. We then recompile and repeat this loop, keeping only patches that pass type checking. We cap the repair loop at 10 turns per target. In our observations, most successful repairs are completed within 3 turns on average. Combined with heuristic search and witness type synthesis, this stage increases the instantiation success rate to around 97%.

It is worth noting that prior work [1] on fuzzing Rust libraries can also generate driver programs, but typically constructs concrete input values for each function parameter to drive execution. In our setting, however, execution is unnecessary; we only need to trigger compilation and materialize machine code. By avoiding the generation of concrete argument values, our approach is both simpler and more efficient.

2.2 Generating Equivalent Pairs

To construct the equivalent (EQ) subset of the benchmark, we compile each source revision under a range of realistic settings, varying the following build configurations.

(1) Toolchain version. We vary compiler version using stable `rustc` releases from 1.72.0 to 1.94.0 (August 2023–March 2026), plus one nightly snapshot dated 15 March 2026. This models divergence caused by compiler evolution during independent rebuilds. We exclude substantially older releases because many crates in our corpus enforce a minimum supported Rust version.

(2) Code generation backend. We vary the code generation backend between LLVM and Cranelift where backend selection is available. LLVM is the default production backend in most Rust pipelines, while Cranelift is increasingly used for fast compilation workflows. These backends differ in instruction selection, register allocation strategy, and optimisation heuristics.

(3) Code generation units. We vary the number of code generation units via Cargo configuration (i.e., `release.codegen-units`). This setting controls how a crate is partitioned for LLVM code generation. Higher values usually improve compile-time parallelism but reduce optimisation scope and inlining opportunities. We evaluate two settings: 16 (the default in Cargo's `release` profile) and 1 (a common setting when users prioritize final artifact quality over compile time).

(4) Link-time optimisation (LTO). We vary whether link-time optimisation (LTO) is enabled. Under LTO, crates are emitted in LLVM IR rather than as fully lowered object files, and the linker invokes LLVM to perform whole-program optimisation before producing the final artifact. LTO can change inlining, specialization, and function elimination decisions, which reshapes layout and symbol structure.

(5) Debug information. We vary whether debug information is preserved or stripped in release builds. In practice, some pipelines keep debug symbols to support diagnosis and observability, while others strip them to reduce artifact size and information exposure. This choice changes metadata sections and symbol records without changing program semantics.

(6) Post-optimization. We optionally apply a post-link optimization step to the final binary artifact. This stage performs tool-driven cleanup and size-oriented transformations after linking. It is enabled only on platforms where suitable tooling is available. For example, on WebAssembly targets we apply `wasm-opt` passes such as `-coalesce-locals` and `-reorder-locals`.

2.3 Constructing Non-Equivalent Pairs

We construct the non-equivalence portion of the benchmark using two complementary sources of semantics-changing variation: (i) AST-guided mutation of a fixed revision, and (ii) API breaking changes mined from real-world crate evolution. In both cases, we compile the two candidate revisions under the same build configuration and label the resulting artifact pair as non-equivalent, retaining only pairs that successfully type-check and compile to our target.

(i) AST-guided mutation of a fixed revision. Starting from a fixed source revision, we generate candidate mutants via source-to-source AST rewrites using Cargo-mutant[3] and our custom plugins.

From Cargo-mutant, we adopt several standard transformations, including return-value substitution (e.g., replacing function bodies with constants, defaults, or wrapper variants such as `Some`/`None` and `Ok`), operator mutations (e.g., swapping comparison, logical, arithmetic, and bitwise operators, or removing unary operators),

[3] https://github.com/sourcefrog/cargo-mutant

match-expression perturbations (e.g., removing match arms when a wildcard is present or rewriting guards to constant `true`/`false`), and struct literal field deletion (removing explicitly specified fields so they fall back to base values).

In addition, we introduce two extensions. First, we perform unsafe pointer mutations by inserting small `unsafe` blocks, replacing selected `&T`/`&mut T` references with raw pointers (`*const T`/`*mut T`), and accessing values through pointer dereferencing. This weakens Rust's safety guarantees and, for non-`Copy` types, alters when or whether the `drop` function is called. Second, we rewrite selected parameter or return types from `T` to `&'a T` or `&'a mut T`, converting owned values into borrowed references. This changes the function's semantics: instead of operating on its own data, the function now operates on data owned by the caller. As a result, changes made inside the function can affect the caller's data.

(ii) Commit-history mining of API-breaking changes. To complement synthetic mutations, we also generate NEQ pairs from real-world crate evolution by traversing each crate's commit history and identifying revision pairs that introduce breaking changes to the public API. For each candidate pair, we extract the set of publicly reachable items (e.g., `pub` functions, methods, types, traits, and modules) using Cargo-public-api[4], and compare their interface signatures across revisions. We classify a pair as API-breaking when we observe removals or renames of public symbols; changes to function or method signatures (including parameter or return types, arity, generic parameters, trait bounds, lifetimes, or `where` clauses); visibility changes (e.g., `pub` to non-`pub`); or structural changes in type definitions (e.g., added or removed fields in `pub` structs, modified enum variants, altered trait method sets, or changes in `impl`/`trait` relationships). To reduce noise from private refactorings, we restrict our analysis to interface-level changes and prioritize adjacent commits with minimal unrelated modifications.

2.4 Dataset Statistics and API Access

Our dataset contains 31,026 crates. For each crate, we compile across major target operating-system platforms that reflect common deployment settings. Where a platform supports multiple runtime environments or architectures, we include all supported target variants. Specifically, the targets include Windows (*x86_64-pc-windows-msvc*, *x86_64-pc-windows-gnu*, *i686-pc-windows-gnu*), Linux (*x86_64-unknown-linux-gnu*, *aarch64-unknown-linux-gnu*, *riscv64gc-unknown-linux-gnu*), WebAssembly (*wasm32-unknown-unknown*, *wasm32-wasip1*, *wasm32-wasip2*), and macOS (*x86_64-apple-darwin*, *aarch64-apple-darwin*). For each target variant, we consider 24 toolchain versions, 2 code-generation backends, 2 code-generation-unit settings, 2 LTO settings, 2 debug-information settings, and 2 post-optimization settings, yielding 768 possible configuration combinations. For the NEQ portion, we generate up to 16 non-equivalent pairs per crate (12 AST mutations and 4 API-change-based pairs, when available) for each configuration; this cap of 16 is chosen mainly due to our limited computational resources.

It should be noted that not all combinations are compatible with every target platform; for example, WebAssembly and *i686-pc-windows-gnu* do not support Cranelift backend, and post-link optimization is applied only to WebAssembly targets. In addition, some crates can only be compiled starting from a minimum toolchain version and may support only a subset of target platforms. For instance, crates that depend on x86_64-specific features such as AVX SIMD intrinsics can be built only for x86_64 targets. In practice, after filtering out unsupported configurations and failed builds, the final dataset contains 19,184,671 EQ artifacts and 273,848,531 NEQ artificats. All builds were executed on a dedicated environment with 2,TB RAM and 240 CPU cores (`Intel(R) Xeon(R) Platinum 8580`), and the full build process took 37 days.

[4]https://github.com/Enselic/cargo-public-api

Listing 2: Two-sided sampling via the benchmark API (illustrative).

```python
# Configure each side requirement independently
lhs = BuildConfig(toolchain="1.92.0")
rhs = BuildConfig(toolchain="nightly-2026-01-15", lto=True)

# Sample an equivalent (EQ) pair under (lhs, rhs).
eq = client.sample(kind="EQ", lhs=lhs, rhs=rhs)
a_bytes, b_bytes = eq.lhs.bytes, eq.rhs.bytes  # raw Wasm bytes

# Sample a non-equivalent (NEQ) pair under (lhs, rhs).
neq = client.sample(kind="NEQ", lhs=lhs, rhs=rhs)
c_bytes, d_bytes = neq.lhs.bytes, neq.rhs.bytes  # raw Wasm bytes
diff_label = neq.diff_patch  # change label (NEQ only)
```

To facilitate dataset navigation, we expose a programmatic API that accepts two independent compiler/build configurations (one per artifact in a pair) and returns the corresponding artifacts together with complete build metadata, as illustrated in Listing 2. For NEQ samples, the API additionally returns a change label in the form of a source-level diff (or patch summary) describing the semantic edit between the paired inputs.

3 Conclusion and Outlook

We present RustBuildEq, a benchmark of EQ and NEQ Rust binary pairs derived from widely used crates under realistic build variability. We are already experimenting with models such as decision trees over machine code, as well as LLM-based techniques to canonicalize and normalize build-induced differences. Looking ahead, we envision RustBuildEq enabling more autonomous and scalable binary equivalence solutions, where AI systems assist or directly perform equivalence reasoning within modern software supply-chain workflows.

References

[1] Georgios Androutsopoulos and Antonio Bianchi. 2025. deepSURF: Detecting Memory Safety Vulnerabilities in Rust Through Fuzzing LLM-Augmented Harnesses. *arXiv preprint arXiv:2506.15648* (2025).

[2] Giacomo Benedetti, Oreofe Solarin, Courtney Miller, Greg Tystahl, William Enck, Christian Kästner, Alexandros Kapravelos, Alessio Merlo, and Luca Verderame. 2025. An Empirical Study on Reproducible Packaging in Open-Source Ecosystems. In *Proc. ICSE'25*. ACM/IEEE.

[3] Jens Dietrich, Tim White, Mohammad Mahdi Abdollahpour, Elliott Wen, and Behnaz Hassanshahi. 2024. Bineq-a benchmark of compiled java programs to assess alternative builds. In *Proc. SCORED'24*.

[4] Jens Dietrich, Tim White, Behnaz Hassanshahi, and Paddy Krishnan. 2025. Levels of binary equivalence for the comparison of binaries from alternative builds. In *Proc. ICSME'25*. IEEE.

[5] OpenSSF. [n. d.]. Reproducible Build | OpenSSF Glossary. https://glossary.openssf.org/reproducible-build/. Accessed 2026-03-13.

[6] OpenSSF. 2023. Supply-chain Levels for Software Artifacts (SLSA). https://slsa.dev.

[7] Reproducible Builds. [n. d.]. Definitions. https://reproducible-builds.org/docs/definition/. Accessed 2026-03-13.