Oxidizer: Toward Concise and High-fidelity Rust Decompilation

Yibo Liu*, Zion Leonahenahe Basque*, Arvind S Raj*, Chavin Udomwongsa*,
Chang Zhu*, Jie Hu*, Changyu Zhao†, Fangzhou Dong*,
Adam Doupé*, Tiffany Bao*, Yan Shoshitaishvili*, Ruoyu Wang*
*Arizona State University
{yiboliu, zbasque, arvindsraj, cudomwon, czhu62, jiehu12, bonniedong, doupe, tbao, yans, fishw}@asu.edu
†Stanford University
changyuz@stanford.edu

***Abstract*—Rust is an increasingly popular language that has gained traction among developers. As a memory-safe language, Rust reduces the burden for developers to create reliable and fast software. However, the same features can also hinder reverse engineering tasks. For instance, malware developers have also picked up on the trend of Rust, using it to make their malware more reliable and difficult to analyze.**

Reverse engineering tasks often rely on decompilers to recover the source code from these binaries. However, analysts find it difficult to analyze Rust binaries using modern C decompilers. Modern C decompilers fail on Rust binaries because they fail to recover high-level Rust abstractions from low-level implementations. As a result, the decompiled output is often verbose and inaccurate. Therefore, we believe that to achieve high-quality Rust decompilation, a decompiler must bridge the gap between high-level Rust abstractions and low-level implementations.

In this paper, we study how C decompilers fail at decompiling Rust binaries. We identify a comprehensive list of decompilation failures, find the root causes of these failures, and develop a novel decompiler, OXIDIZER, for decompiling Rust binaries to Rust pseudocode. We evaluate OXIDIZER on 28 popular Rust projects across multiple optimization levels and compiler versions, comparing it against angr, Hex-Rays, Ghidra, and Binary Ninja. OXIDIZER outperforms all baselines on most conciseness and fidelity metrics, and is the only tool capable of recovering Rust enums and macros. A human study further shows that participants using OXIDIZER achieved 28% higher accuracy and completed tasks 20% faster than those using Hex-Rays.

1. Introduction

The Rust programming language [1] has grown in popularity and is used for critical systems such as embedded firmware [2] and the Linux kernel [3]. This adoption has largely been driven by Rust's memory-safety guarantees, reducing the burden of building reliable and performant software [4]. On the other hand, malware authors have also adopted Rust to build malware for its high execution speed, cross-platform capabilities, and resistance to reverse engineering [5].

Recent advancements in binary decompilation [6–8] have facilitated its application in security tasks such as malware analysis [9]. However, most modern decompilers are primarily designed for C/C++ binaries and do not account for Rust's characteristics [6–8, 10, 11]. Given the growing prevalence of Rust-based malware, this design gap makes existing decompilers ineffective for analyzing Rust binaries (as shown in Section 2). One trivial workaround is to apply post-decompilation fixes on C-oriented decompilers' output, using methods such as case-specific scripting [12, 13] and manual rewriting [13]. Even though there is no public work on using LLMs to decompile Rust binaries, we found that recent research has explored LLM-assisted C decompilation [14–17]. We experimented with trivial post-decompilation fixes by prompting LLMs, but found that there are some fundamental issues with LLM-based post-decompilation fixes, hallucination and information loss, which we will explain in Section 2. Therefore, a dedicated Rust decompiler is still needed. LLM-based post-decompilation fixes tend to hallucinate, producing code that appears coherent but lacks semantic grounding, while scripting and manual rewriting can only address trivial issues such as correcting Rust string literals or demangling function names. More complex challenges, such as reconstructing Rust enums or recovering Rust-specific control-flow constructs, remain largely unsolved.

In this paper, we first examine how C-oriented decompilers perform on Rust binaries. We evaluate four popular C decompilers on 28 popular Rust projects on GitHub. Based on the evaluation, we find that decompiling Rust binaries into C pseudocode often yields verbose code with reduced abstraction, loss of semantic information, and potential inaccuracies. We compile a comprehensive list of *fidelity issues* observed in these decompilation outputs and trace their root causes to two Rust-specific challenges: (1) Rust introduces high-level abstractions that have no direct counterparts in C (e.g., enums, pattern matching); (2) Even for similar abstractions, the Rust and C compilers generate different low-level implementations (e.g., string handling and library linkage).

We then develop novel techniques to address the fidelity issues in Rust decompilation and implement them in OXIDIZER, a prototype Rust decompiler built on top of angr [18]. OXIDIZER performs binary-level analyses to overcome challenges introduced by the Rust compiler, enabling Rust standard library function identification and the application of known struct and function types. OXIDIZER also addresses identified fidelity issues by designing the three core components of decompilation, control-flow recovery, type recovery, and structuring, specifically for Rust. With these Rust-oriented designs, OXIDIZER is able to recover Rust high-level abstractions such as macros, enums, and pattern matching. OXIDIZER takes a first step toward high-quality Rust decompilation by generating concise, faithful, and human-readable code that enables more effective reverse engineering of real-world Rust binaries.

Finally, we evaluate OXIDIZER on our dataset consisting of 27 projects from the top 50 Rust projects on GitHub (ranked by stars) and the Rust reimplementation of GNU Coreutils [19], with various optimization levels and two different Rust compiler versions. OXIDIZER generates output that is significantly more concise and faithful to the original program compared to existing binary decompilers. It produces 15% fewer lines of code and 7% lower cyclomatic complexity than the best-performing C decompiler on average. Moreover, OXIDIZER effectively reduces extraneous function calls and achieves the highest matched macro recovery rate among all evaluated decompilers. For type inference, OXIDIZER outperforms other decompilers on recovering struct and enum types, which are the most prevalent types in our dataset. In our user study with 37 participants of different reverse engineering expertise, participants using OXIDIZER achieved 28% higher task scores and completed tasks 20% faster than with Hex-Rays, demonstrating improved efficiency and accuracy. Participants also rated OXIDIZER more favorably, with an average score of 4.49 out of 5, compared to 2.61 for Hex-Rays.

Contributions. We make the following contributions:

1) We empirically study the performance of state-of-the-art C decompilers on Rust binaries, identifying key *fidelity issues* and Rust-specific challenges that hinder effective decompilation.
2) We propose the first Rust decompilation pipeline to address these challenges and implement it in our open-source prototype, OXIDIZER, providing a foundation for future research in Rust decompilation.
3) We introduce the first systematic methodology for evaluating Rust decompilation in terms of *conciseness* and *fidelity*, and use it to benchmark OXIDIZER against state-of-the-art C decompilers, showing its superior performance. We further demonstrate OXIDIZER's practical benefits through case studies on real-world malware samples and a human study involving reverse engineering tasks.

To further open science, we release OXIDIZER and all our evaluation artifacts at https://github.com/sefcom/oxidizer and integrate OXIDIZER into angr.

2. Motivation

To understand the challenges of Rust decompilation from a technical perspective, we reviewed malware analysis reports in `rust-malware-gallery` [13] and collected analysts' complaints. Most complaints simply note that analyzing Rust malware is difficult, without explaining the specific challenges. A few more detailed complaints point to issues such as mangled function names, extraneous code inserted during compilation for security checks or resource management, non-null-terminated strings, and inlined Rust standard library functions. While some issues, like non-null-terminated strings, can potentially be addressed with post-decompilation fixes, others require fundamental changes to decompiler internals to resolve.

To illustrate these issues concretely, we present a motivating example from a real-world Rust project [20]. Listing 2 shows (a) the original Rust source code of function `encrypt_file`, alongside (d) the decompiled C pseudocode generated by the state-of-the-art commercial decompiler Hex-Rays [10], and (b) part of the Rust pseudocode generated by Binary Ninja [21]. For comparison, we also include (c) the result of a post-decompilation fix using ChatGPT [22] based on Hex-Rays' output.

Comparing to the source code, Hex-Rays' decompiled C pseudocode exhibits several critical issues (highlighted in red boxes): (1) Inlined `Vec::new` function call; (2) Expanded `eprintln` macro call; (3) Missing format string; (4) Redundant code caused by the inclusion of resource release function calls (i.e., `drop_in_place`). Green boxes highlight the corresponding elements in the original source code. Though Binary Ninja produces Rust pseudocode, it suffers from the same issues as Hex-Rays.

In recent years, researchers have explored using LLMs to improve C decompilation quality [14–17]. However, there is no public work on using LLMs to decompile Rust binaries. We experimented with trivial post-decompilation fixes by prompting LLMs, but found the output had several issues. As shown in Listing 2 (c), using ChatGPT to recover Rust code from Hex-Rays' output introduces fake function signatures, fake function calls, and fake strings (highlighted in purple boxes) that do not exist in the original source, leading to significant fidelity issues. Other LLMs we experimented with have similar issues. Moreover, LLM-based post-decompilation fixes face fundamental limitations. The reason is twofold: 1) the decompilation process inevitably involves information loss when lifting low-level code to higher-level representations, and 2) all LLMs are susceptible to hallucinations to some degree [23].

These observations highlight the need for a decompiler that is explicitly designed to handle Rust-specific compilation patterns and abstractions, rather than adapting C-oriented decompilers or relying on post-decompilation fixes. To that end, we empirically study the fidelity issues that arise during the decompilation of Rust binaries, identify their root causes, and propose solutions to address them. We implement our solutions in OXIDIZER, a Rust decompiler.

```
fn FakeCrypt::fileops::encrypt_file(a0: &Path, a1: i64, a2: i64) {
  let v0: I32NotAllOnes;  // [bp-0x818]
  let v2: Result<struct4, struct8>;  // [bp-0x810]
  let v3: Vec<u8, alloc::alloc::Global>;  // [bp-0x800]
  let v4: u64;  // [bp-0x7f0]
  let v5: File;  // [bp-0x7d8]
  let v6: Result<File, Error>;  // [bp-0x7b8]
  let v7: File;  // [bp-0x7b4]
  let v8: struct960;  // [bp-0x3f8]
  let v11: u64;  // rax
  let v12: u64;  // rdx
  let v13: Result<(), &BOT>;  // rax
  let v14: Result<usize, Error>;  // rax:rdx

  v2 = std::fs::File::open(a0);
  match v2 {
    Err(_) => {
      return;
    },
    Ok(v0) => {
      v3 = Vec::new();
      v14 = <std::fs::File as std::io::Read>::read_to_end(&v0, &v3);
      if let Err(_) = v14 {
        return;
      }
      v3 = alloc::vec::Vec<T,A>::resize((9223372036854775792 & v4) + 16, 0);
      v6 = <aes::autodetect::Aes256 as crypto_common::KeyInit>::new(a1);
      v8 = <cbc::decrypt::Decryptor<C> as
      ↪  crypto_common::InnerIvInit>::inner_iv_init(&v6, a2);
      v11 = cipher::block::BlockEncryptMut::encrypt_padded_mut(&v8, 1, v4,
      ↪  v4);
      if v11 {
        v6 = std::fs::File::create(a0);
        if let Ok(v5) = v6 {
            v13 = std::io::Write::write_all(&v5 as u64, v11, v12);
        }
      } else {
        eprintln!("[!] Encryption failed for {}: {}", &a0, &v1);
      }
      return;
    },
  }
}
```

Listing 1: OXIDIZER decompilation of the motivating example (41 lines of Rust pseudocode).

Listing 1 shows OXIDIZER's decompilation of the same function. Compared with Hex-Rays, Binary Ninja, and ChatGPT-based fixes, OXIDIZER produces results that most closely resemble the original source, successfully recovering Rust-specific high-level abstractions such as type information, control-flow constructs, and correct semantics. We detail the techniques implemented in OXIDIZER in Section 5.

3. Background and Research Scope

Rust and Rust Malware. Rust ensures safety through compile-time lifetime checks, runtime bounds checks, and an ownership model that prevents data races. These safety features and a rich and powerful type and trait system make Rust an attractive choice for writing safe yet expressive programs [24].

Malware authors have also begun adopting Rust (instead of traditional choices such as C, C++, or Delphi) to develop malware [13]. In addition to the various language features, easy cross-platform support and the lack of tools to efficiently analyze Rust binaries have made Rust an attractive choice for malware [12]. Prevalent malware, such as the HIVE ransomware and the WildCard ransomware, have been rewritten in Rust in recent years [25, 26].

C Decompilers on Rust Binaries. The Rust compiler `rustc` uses LLVM as its backend [27], producing native machine code that existing C decompilers can disassemble and lift without Rust-specific support. However, these decompilers lack Rust ABI knowledge and misapply C-centric heuristics, misinterpreting Rust-specific abstractions (e.g., enum layouts, pattern matching) that have no direct C equivalent. Even for shared abstractions like strings and function calls, `rustc` and C compilers generate different low-level implementations, further degrading decompilation quality.

Rust Support in Modern Decompilers. To the best of our knowledge, there is no specialized Rust decompiler publicly available. Modern decompilers have primarily focused on optimizing each of these stages for C- and C++-specific output [28]. Analysts often use state-of-the-art C decompilers, including angr, Hex-Rays, Ghidra, and Binary Ninja, to analyze Rust binaries. These decompilers only have limited support for Rust decompilation. They all support Rust-style function name demangling. Binary Ninja offers a Pseudo-Rust representation since 4.2 [29], which displays decompiled code in Rust-like syntax.

Analysts attempted various post-decompilation fixes to generate Rust pseudocode from C decompilation output. Ghidrust [30] is a Rust binary analysis extension for Ghidra, which provides functionalities including Rust binary detection, Rust standard library function detection, and emitting Rust pseudocode by parsing decompiled C code. However, as we will show later, these solutions are insufficient for generating meaningful Rust pseudocode for reverse engineering.

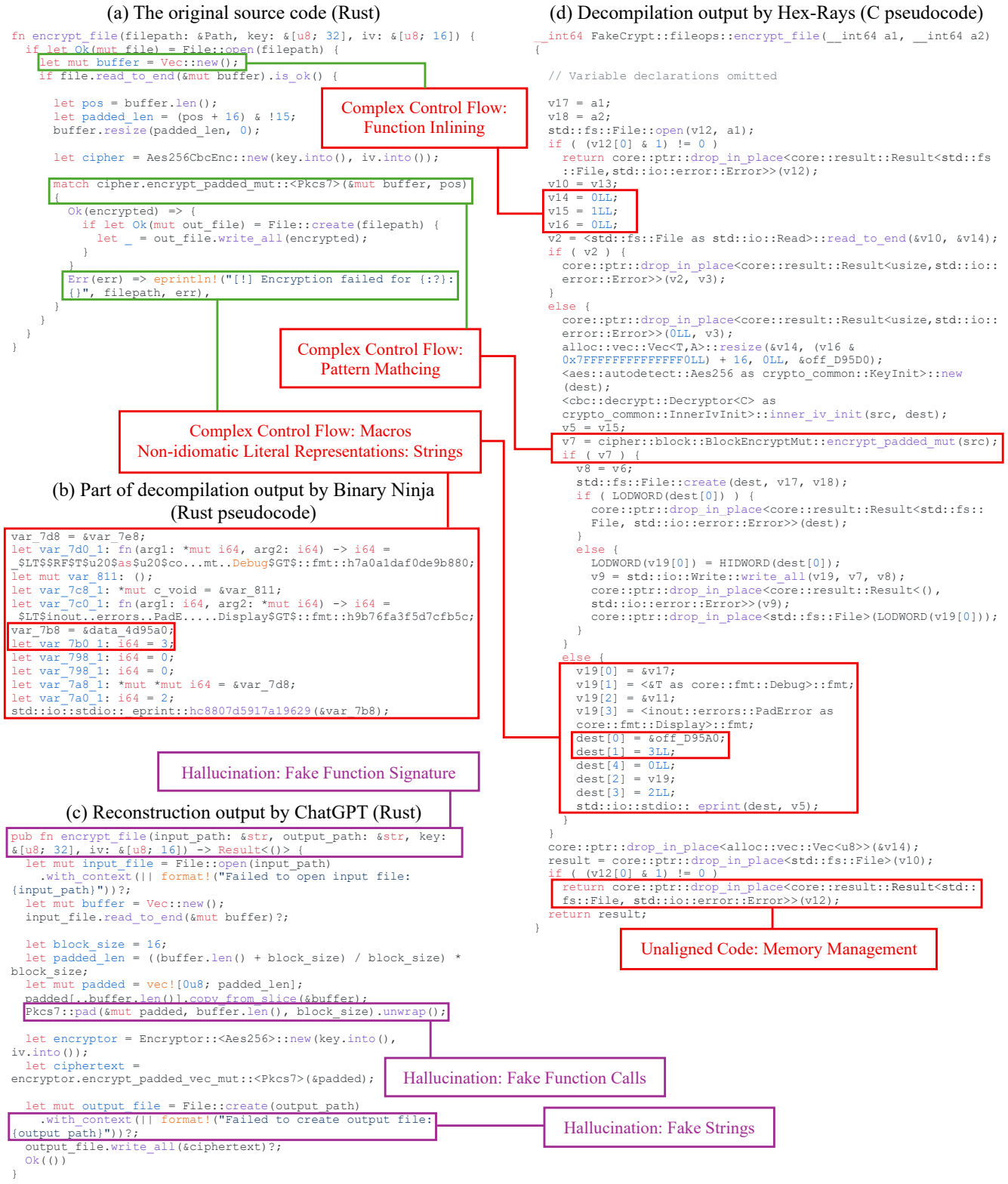
Research Scope. We design OXIDIZER to assist with reverse engineering tasks on Rust binaries, including, but not limited to, Rust malware analysis. While designing OXIDIZER, we had to make trade-offs in the metrics we prioritized for our decompilation. For instance, improving the conciseness of our decompiler, such as removing redundant dealloc calls, can also potentially remove security-relevant information. Our goal with OXIDIZER is to generate decompilation that is *readable* and intended for human use. As such, we prioritize closeness to source code (Rust), which previous work has associated with readability [31]. Improving tasks such as recompilability and robustness to obfuscation is outside the scope of our design, and we leave them to future researchers.

Modern C decompilers (e.g., Hex-Rays) do not attempt to generate recompilable or fully correct C code. This is a feature for malware analysis—from a participant in a human study on malware analysis "*It is useful to get a snippet of code from malware to use externally. For that, function correctness is important, but whole code correctness is not*" [9]. Likewise, we do not aim to generate recompilable Rust code.

We expect as input an unaltered executable generated by the Rust compiler. When facing adversarial binaries, such as those generated by packing [32–34] or obfuscation [35–39], analysts must first unpack or deobfuscate before decompiling them, just as they use state-of-the-art C decompilers. Deobfuscation or unpacking is out of scope. OXIDIZER supports decompiling stripped Rust binaries.

4. Decompilation Fidelity Issues

To study the fidelity issues in Rust binary decompilation, we evaluate four popular decompilers (Hex-Rays,



Listing 2: The original Rust source code (23 lines, a), part of Binary Ninja's decompilation (12 lines of Rust pseudocode, b), Hex-Rays' decompilation (81 lines of C pseudocode, d) of the same function, and ChatGPT-reconstructed code based on Hex-Rays decompilation (21 lines, c). We omitted variable declarations from the decompiled code for brevity.

Ghidra, Binary Ninja, and angr) on 28 popular Rust projects from GitHub, compiled under various optimization levels and compiler versions. To enable one-to-one function-level matching between source code and binaries, we disabled inlining during compilation[1].

Researchers have discussed fidelity issues in C decompilers [31], and we adopt a similar concept in this paper for Rust decompilation, where fidelity includes both readability and correctness issues that prevent the decompiled output from faithfully reflecting the original program. While traditional C decompilation has its own challenges, in this paper, we focus on Rust-specific fidelity issues.

Overall, existing decompilers perform insufficiently on Rust binaries because they fail to faithfully recover high-level Rust abstractions by design. Most modern decompilers are built for C/C++ binaries and thus are optimized to reconstruct C/C++-specific abstractions. While Rust and C/C++ share some common constructs, they also introduce unique abstractions without direct counterparts in the other. Moreover, Rust and C/C++ compilers generate different low-level implementations for similar high-level concepts that existing C/C++ decompilers are not designed to handle.

Fidelity Issue 1: Enumeration and Pattern Matching. In Rust, enumeration (enum for short) is a sum type that can hold one of several named variants with distinct names and fields. A common operation associated with Rust enums is pattern matching, which is implemented using the `match` control-flow construct. This construct branches execution based on the specific variant an enum holds. In particular, Rust uses two built-in enums, `Option<T>` and `Result<T, E>`, to handle recoverable errors. Instead of relying on exceptions, Rust encourages explicit error handling using pattern matching.

Issue 1.1: Memory Layout Abuse. The difference between C's `switch-case` and Rust's pattern matching is that Rust's pattern matching checks if the instance matches a specific variant instead of value. Each Rust enum instance consists of a discriminant, an integer that indicates which variant it currently holds, and shared space for variants' fields [40]. In decompiled C pseudocode, enum initialization and variant field access typically appear as a sequence of raw memory writes. While this reflects low-level implementations, it loses high-level semantics such as the specific variant being constructed. Likewise, pattern matching on enums is often compiled into a series of if-else checks on the discriminant accessed through raw memory reads.

Issue 1.2: Complex Control Flow. Existing C decompilers recover Rust pattern matches as `if-else` structures, which are functionally equivalent but not idiomatic. The `if-else` structures lack high-level semantics, including variant names and associated fields, making the recovered code less readable and harder to understand. Listing 2 shows an example of complex control flow caused by pattern matching.

Fidelity Issue 2: Error Propagation. Rust provides a convenient syntax for error handling using the question mark operator (`?`). This operator is appended to an expression of `Result<T, E>` type. It simplifies error propagation by implicitly handling `Result` values: if the result is an `Err`, it returns early from the function; otherwise, it unwraps the `Ok` value.

During compilation, the `?` syntactic sugar is desugared into more verbose control flow, typically a pattern matching construct that returns early on the `Err` variant. This pattern matching construct is further compiled into low-level implementations, as described in Fidelity Issue 1.2.

Fidelity Issue 3: Macros. Rust enables powerful metaprogramming through three types of macros: function-like, attribute-like, and derive macros. The most commonly used are function-like macros, such as `println!` and `panic!`. Function-like macros enable pattern-based code generation, which is similar to C-style macros but more expressive. These macros match the input token stream against defined patterns and expand the input into corresponding Rust code during compilation. The effect of macros on decompilation is similar to function inlining: a concise macro call in the source code can expand into a larger, less readable chunk of code during compilation.

Fidelity Issue 4: Deref Coercion. Deref coercion is a Rust feature that enables implicit reference type conversion for types that implement the `Deref` trait. Specifically, if a type `T` implements `Deref<Target = U>`, a reference of type `&T` can be automatically converted to a reference of type `&U`. This allows values of type `T` to access methods or fields defined on `U` without explicit conversion.

Issue 4.1: Unaligned Code. To support this feature, the Rust compiler inserts calls to the appropriate `Deref::deref` implementation at compile time. Although implicit in the source code, these deref calls appear explicitly in the compiled binary. Listing 3 shows an example of complex control flow caused by deref coercion.

Issue 4.2: Memory Layout Abuse. When deref coercion is compiled into explicit `deref` function calls and subsequently inlined, they might be compiled into raw memory accesses. Listing 4 shows an example of implicit conversion between `&String` and `&str`, and what it looks like in Hex-Rays' C pseudocode.

(a) Rust source code.

```
s = String::from_utf8_lossy(&stderr).to_string();
```

(b) C pseudocode as decompiled by Hex-Rays.

```
String::from_utf8_lossy(&v17, v26, v27);
v11 = <Cow<B> as Deref>::deref(&v17);
<T as ConvertVec>::to_vec(&v23, v11);
```

Listing 3: The `from_utf8_lossy` function in the source returns a `Cow<str>` instance, while the `to_string` method (inlined to `to_vec` function in Listing 3b) takes a `&str` argument. The `&Cow<str>` instance is converted to a `&str` implicitly by deref coercion. However, the `deref` function call appears explicitly in Hex-Rays decompilation output.

1. Function inlining cannot be completely disabled in Rust.

(a) Rust source code

```
let suffix = format!("{suffix_from_template}");
if suffix.contains(MAIN_SEPARATOR) {
    ...
}
```

(b) C pseudocode as decompiled by Hex-Rays

```
...
core::option::Option<T>::map_or_else(&v69, v47);
if ( !char::is_contained_in(*(&v69 + 1), v70) )
{
    ...
}
```

Listing 4: The `contains` method is defined on `&str` type, while the `suffix` variable has the type `String`. This conversion appears explicitly in Hex-Rays' decompilation output (Listing 4b), by dereferencing the `ptr` and `len` fields (i.e., `*(&v69 + 1)` and `v70`) in the `String` instance (i.e., `v69`) to construct the equivalent string slice.

Fidelity Issue 5: Statically-linked Standard Library Functions. Most C compilers by default link the standard library dynamically into the binary [41, 42]. By contrast, in Rust, the standard library and dependencies are statically linked into the final binary by default [43]. It is also uncommon and not encouraged to dynamically link the Rust standard library due to unstable ABI across Rust versions, deployment complexity, and community consensus [44]. Because the Rust standard library is typically statically linked into the binary, its function symbols can be stripped during compilation or post-processing. When stripped, standard library functions become indistinguishable from user-defined ones, forcing reverse engineers to identify and recover their original functionalities. Additionally, statically linked standard library and third-party library functions enable the Rust compiler to inline them. It is harder for reverse engineers to understand the context with inlined function definition than the function name which summarizes the functionality [31].

Fidelity Issue 6: Resource Management. In C, programmers must explicitly create and release resources such as heap memory, file descriptors, and network sockets [45]. In contrast, Rust uses the ownership model with a set of rules to manage resources safely and automatically [46]. One key rule of ownership is that when a variable goes out of scope, the resources it holds are automatically released. Rust programmers do not need to explicitly release memory or other resources. Instead, the compiler inserts appropriate low-level code, such as calls to destructors, at compile time. The compiler-inserted resource cleanup code does not align with any code in the original source. During reverse engineering, we found that such code frequently appears in the decompilation.

Fidelity Issue 7: Security Checks. Besides resource management, there are also security properties that are difficult to check at compile time (e.g., indexing out of bounds and dividing by zero). Rust compiler inserts security-check code that checks these security properties at runtime. Similar to resource management, implicit security checks also introduce extraneous code that does not exist in the source.

(a) Decompilation with correct struct layout

```
v5 = Arguments {
    pieces: ["Source file does not exist: "]
    args: [v0]
    fmt: None
};
v3 = alloc::fmt::format::format_inner(&v5);
```

(b) Decompilation with incorrect struct layout

```
v6 = Arguments {
    pieces: ["Source file does not exist: "]
    fmt: &v0
    args: &[Argument] {
        ptr: 0
        len: <UNKNOWN>
    }
};
v4 = alloc::fmt::format::format_inner(&v6);
```

Listing 5: In this case, the correct field order of `Arguments` is `pieces`, `args`, and `fmt`, which does not align with the field order in the source. OXIDIZER is not able to recover the correct definition of `Arguments` with the wrong field order, as Listing 5b shows.

Fidelity Issue 8: Rust String Literals. In C, ANSI strings are null-terminated, with the terminating byte indicating the end of the string [47]. A Rust string literal consists of a pointer to the character sequence, typically not null-terminated, and an integer representing its length [48]. Most existing decompilers do not support Rust string literals, which results in Rust string literals being represented as regular global variables in decompiled code. Interestingly, when used as macro arguments in Rust, string literals are not simply expanded as they are in C macros, instead, they are used as a token stream, which can be manipulated during expansion (e.g., a format string is divided into several pieces as an array). Failing to recover macro calls may also lead to this fidelity issue.

Fidelity Issue 9: Reordered Struct Layout. In C, within a struct object, addresses of its fields increase in the order in which the fields were defined [49]. However, the memory layout of a Rust struct, that is, the order of its fields is not guaranteed to match the order in the source code [50]. While field reordering itself does not directly cause fidelity issues, it can hinder decompilers from using known Rust standard library structs and recovering Rust high-level abstractions. For example, recovering a struct instance relies on matching field types to memory offsets to group field assignments into a single, unified initialization statement. If field types and offsets are mismatched due to incorrect assumptions about layout, the recovered struct will be incorrect. Listing 5 shows an example of this issue.

Fidelity Issue 10: Unstable Enum Variant Discriminants. Rust allows custom discriminant values to be assigned to enum variants. However, unless explicitly specified, these values are not guaranteed to be stable across different compiler versions [40]. By default, the compiler typically assigns discriminant values starting from zero and incrementing by one, but this is not guaranteed for the purpose of optimization. This instability can pose challenges to decompilers, as

correct discriminants are fundamental to recovering pattern matching constructs.

5. Rust Decompilation

We design a Rust decompiler, OXIDIZER, as shown in Figure 1. Given a Rust binary, OXIDIZER begins with control-flow graph (CFG) generation and Rust-specific *Binary-level Analysis*, identifying the Rust compiler version and locating standard library functions; it then loads the corresponding struct and function prototype database. For each function, OXIDIZER first generates the function-level CFG (fCFG), and performs *fCFG Simplification without Types* to remove resource-release and safety-check code. It then conducts *Rust Type Inference*, recovering variable locations, function prototypes, and local variable types, including structs and enums. With these types, OXIDIZER applies *fCFG Simplification with Types* to further refine the control flow. In the *Structuring* phase, OXIDIZER converts the fCFG into high-level control-flow constructs such as if-else and loops, as well as Rust-specific constructs such as pattern matching and error propagation. Finally, in the *Rust Pseudocode Generation* stage, OXIDIZER outputs structured Rust-like pseudocode.

In detail, OXIDIZER employs angr for CFG and fCFG construction. After the two steps, OXIDIZER diverges from angr decompilation by its novel Rust-specific simplification, type inference, and control-flow structuring tailored to Rust semantics, which enables the recovery of Rust-only abstractions absent in traditional C-oriented decompilers.

5.1. Binary-level Analysis

Rust Compiler Version Identification. During binary-level analysis, OXIDIZER first identifies the Rust compiler version used for the target binary. We achieve this goal using FLIRT [51], which recovers function symbols by converting the bytes of the function into a signature and matching it against a database of known signatures. We build FLIRT signatures for each Rust compiler version from 1.39.0 to 1.93.0, and match the Rust binary with every FLIRT signature file. FLIRT produces the most matches when the Rust binary and the FLIRT signature file are produced from the same version of the Rust compiler. OXIDIZER uses this feature to identify the Rust compiler version.

Rust Standard Library Function Identification. If symbols are not present in the binary, OXIDIZER applies the FLIRT signature file with the highest match rate in the prior stage to identify Rust standard library functions.

FLIRT Signature Propagation. However, FLIRT has two limitations in terms of identifying Rust standard library functions. First, FLIRT identifies known functions using masked function prologue bytes (e.g., function call addresses will be masked), it is unable to distinguish simple functions that only consist of one or several function calls. Second, FLIRT is not able to identify unknown monomorphized generic functions. For example, the function `core::ptr::drop_in_place<T>` may have different implementation for each generic type `T`.

FLIRT fails to identify a large amount of standard library functions, especially resource-release functions, due to these limitations. To address this issue, OXIDIZER propagates FLIRT-recognized function symbols to their caller functions if their caller functions only contain a single call. Additionally, if an unmatched function contains calls *only* to resource-release functions (identified by FLIRT), we mark that function as a resource-release function as well. This process is recursively applied until a fixed point is reached.

Rust Standard Library Type Application. C decompilers often apply known C library function prototypes and struct definitions to improve decompilation quality. For example, angr parses and embeds thousands of function prototypes from glibc, Win32 APIs, and so on. Likewise, we parse struct definitions and function prototypes from different versions of Rust standard libraries, and embed them into OXIDIZER.

There are two additional challenges to apply struct and function types to Rust binaries: the layout of structs and the discriminants of enum variants are not guaranteed to be the same across different Rust binaries (Section 4, Fidelity Issues 9 & 10). We looked into the Rust compiler source code and found that while Rust does not guarantee the layout of structs and the discriminants of enum variants to be consistent across Rust compiler versions, the logic of struct layout reordering and discriminant assignment is deterministic within the same Rust compiler version, unrelated to what program it is compiling. As such, we build a Rust standard library type database for each Rust compiler version. OXIDIZER loads and applies the corresponding Rust standard library type database for the identified Rust compiler version.

5.2. fCFG-level Simplification without Types

Resource Release Simplification. Rust's automatic resource management feature introduces extraneous complex code in decompilation (Section 4, Fidelity Issue 6). To tackle this issue, OXIDIZER removes Rust-only resource-release function calls and their related code, such as `drop_in_place`, `drop`, and `__rust_dealloc`.

Security Check Simplification. OXIDIZER removes function calls that are related to security checks (e.g., index out of bounds) as well as related code.

5.3. Rust Type Inference

We found that Rust codebases have a high density of function calls (8.19 calls per function with 22.28 LoC on average in our dataset), and struct and enum types are prevalent (there are 16,650 struct types and 7,586 enum types among all 39,868 types in our dataset). However, traditional constraint-based type inference algorithms are mostly intra-procedural, which miss the type information

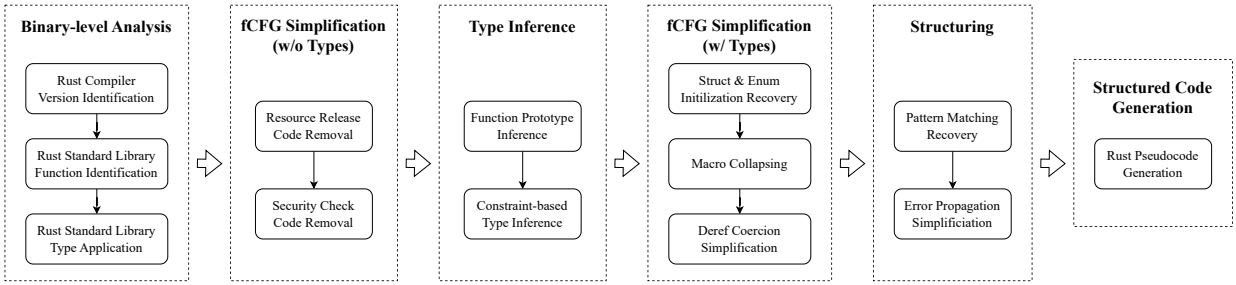


Figure 1: Overview of OXIDIZER Decompilation Pipeline.

from callee functions. They are also very conservative at inferring struct types, and are unable to infer enum types. As such, we implement inter-procedural function prototype inference to mitigate these limitations.

Function Prototype Inference. OXIDIZER infers (1) struct and enum function return types and (2) struct function argument types.

Struct and Enum Return Types. OXIDIZER analyzes both the callee function and its call sites to infer the return type of a callee function. When a Rust function returns a large struct (typically larger than 16 bytes), it writes the struct content to a caller-allocated buffer to which the first function argument point. OXIDIZER traverses all non-looping paths leading to every return block and collects memory writes to the buffer on each path. When writes to multiple offsets in the buffer or writes of multiple sizes exist, it merges the writes on all paths into one to infer the size of the returned struct (or enum, as we will explain next). For structs of 16 bytes or fewer, the Rust compiler may instead return them via registers (e.g., `rax` and `rdx` on x86-64). OXIDIZER handles this case by detecting multi-register returns through calling convention analysis.

In most cases, the sizes of the inferred struct types (memory writes or register writes) along different paths are the same because Rust functions must return a fixed-size type. An exception is enum types where different enum variants may hold associated fields of different sizes. In this case, OXIDIZER infers the returned type as an enum type.

Error Handling Enum Return Types. OXIDIZER also infers error handling enum return types using heuristics. `Result<T, E>` and `Option<T>` are two common enum return types for recoverable error handling. If the inferred type is an enum type with two variants, OXIDIZER checks if the return type could be an error handling enum type using the following heuristics. As Section 4 explains, Rust binaries distinguish enum variants using the discriminant stored at the beginning of the enum value.

The `None` variant of `Option<T>` has no associated fields. When an inferred enum type has two variants and one of them has a discriminant but no associated fields, OXIDIZER infers it as `Option<T>`, treating that variant as `None` and the other (with one associated field) as `Some`. The same heuristics also apply to `Result<T, E>`. To distinguish `Ok` from `Err`, OXIDIZER uses Rust's declaration order: The variant with the lower discriminant value is `Ok`. When niche-value optimization eliminates one discriminant (i.e., only one variant has a discriminant), OXIDIZER distinguishes variants based on context. For `Option<T>`, the missing discriminant indicates the `Some` variant. For `Result<T, E>`, if the matched variant leads to an early return after the call site, OXIDIZER assigns the other variant as `Ok`; otherwise, the larger variant is empirically assigned as `Ok`. Fully niche-optimized types such as `Option<&T>`, where the enum is represented as a single pointer with null indicating `None`, are indistinguishable from raw pointers and integers in binary code and are a limitation of OXIDIZER.

Struct Argument Types. When struct arguments are constructed at call sites or accessed in the callee function, OXIDIZER can infer struct argument types by collecting memory writes and reads.

Constraint-based Type Inference. We implement the constraint-based type inference algorithm Retypd [52] for OXIDIZER and enhance the type system and the solver with enum types. OXIDIZER performs function prototype inference for non-standard-library callee functions and the analyzed function itself. The recovered function argument types and return types will be translated to type constraints, which will significantly contribute to the type inference result.

5.4. fCFG-level Simplification with Types

Struct & Enum Initialization Recovery. OXIDIZER recovers the initialization of struct and enum variables, using the type and variable information recovered by the prior type inference stage.

Outlining Macros. OXIDIZER outlines frequently used standard macros, including `println`, `format`, `write`, `panic`, and so on. Rather than relying on superficial syntactic pattern matching, OXIDIZER performs inter-procedural control-flow and data-flow analysis on the simplified fCFG, combined with recognized Rust standard library function symbols and inferred variable types, to identify macro expansion patterns across different compiler versions and optimization levels.

Deref Coercion Simplification. OXIDIZER identifies explicit struct type conversions that are handled by

`Deref::deref` function calls and reverts them to the original implicit form as in Rust. These function calls are sometimes inlined in Rust binaries. Because deref coercion is implemented differently for different type pairs, the simplification for inlined deref coercion in OXIDIZER must be specified for each type pair. OXIDIZER implements inlined deref coercion simplification for `&String-&str` and `&Vec-slice` pairs.

5.5. Structuring

Pattern Matching Recovery. This component aims to identify potential pattern matching constructs. OXIDIZER traverses all condition nodes and checks if the condition is a comparison between the discriminant of an enum typed value and the expected discriminant. When eligible, OXIDIZER replaces the node with a `match` or `if let` node.

Error Propagation Simplification. After pattern match recovery, OXIDIZER then attempts to find all recovered pattern match structures that have an early return branch. Eligible `match`es will be simplified to the `?` error propagation operators.

6. Decompilation Evaluation Metrics

Existing work evaluates decompilation quality primarily along two dimensions: conciseness [6, 7, 53] and source-code fidelity [8, 31]. Conciseness measures code complexity (e.g., Cyclomatic Complexity [54]), while fidelity reflects how closely the output matches the original source [31]; these two dimensions have been shown to be unaligned [8]. Existing fidelity metrics are either qualitative and manually assessed [31] or tailored specifically to C [8].

Although prior work evaluates decompiled C code, evaluating complexity becomes harder when decompiling non-C languages using C-oriented tools. Some work applies machine-learning metrics such as string edit distance [55], but we find these insufficient for capturing structural aspects such as control flow and types. Therefore, we design new metrics and extend existing ones to assess the conciseness and fidelity for decompiling Rust binaries and especially malware [9, 28, 56].

Conciseness Metrics. We motivate these metrics using previous work by Enders et al. that evaluates the quality of decompilers on malware using human studies [56]. Their work identifies metrics that analysts found important during malware reverse engineering with various decompilers.

Cyclomatic Complexity. Cyclomatic Complexity (CC) [54] is a software engineering metric that quantitatively estimates code complexity and is used in previous decompiler evaluations [53]. CC measures the number of independent paths through a program, which estimates the impact of conditions on control flow. This metric rewards decompilers for reducing the total number of explicit conditions, like booleans in if-statements, which addresses a common complaint of decompiler users reported in previous work [56].

Lines of Code (LoC). LoC is the most common way to evaluate the conciseness of code [6, 7]. It gives an overview of general complexity in a program [57].

Number of Operators (NofOp). To compensate for lines of code full of chained expressions, we also count the total number of operators present. These operators include arithmetic, assignment, comparison, and logical operations. Similarly, previous work uses logical operators to estimate the complexity of decompilation [8]. This metric rewards decompilers that simplify data flow and conditions.

Number of Variables (NofVar). Another reported complaint of decompiler users in previous work [56] is that decompilation often introduces "unnecessary variables." As such, we also evaluate the number of variables present in decompilation, which can estimate a decompiler's ability to coalesce and simplify data access.

Fidelity Metrics. Previous work shows that decompilations closer to their source are more readable [31], and fidelity also serves as a proxy for *correctness*. To assess the fidelity of OXIDIZER on Rust code, we design metrics informed by the challenges in Section 4 as follows.

Gotos. Many previous works [6–8, 53, 58] use the number of goto statements in decompilation as a conciseness metric. However, in Rust source code `goto`s are invalid control-flow structures. To align with source, Rust decompilers should emit zero `goto`s to be correct.

Matched String Literals. String literals are key in software reverse engineering [56]. A good decompiler should preserve the strings found in the original source. Rust binaries notoriously make it harder to display string literals in decompilation. This metric measures the fidelity of data displayed in decompilation.

Matched Source Function Calls. Since Rust binaries contain many inlining and transformations, an ideal Rust decompiler must have the same number of each call found in the source. This metric measures control-flow fidelity as used in prior work [8].

Extraneous Function Calls. Rust compilers introduce extraneous function calls that do not align with any function calls in the source. An ideal Rust decompiler should remove these extraneous function calls. This metric measures control-flow fidelity similar to the previous metric.

Matched Source Macro Calls. Like function calls, macro calls also indicate how well a decompiler is at recovering original source control flow. In Rust, macros like `println` are often used. High-fidelity Rust decompilation should match as many of these macros as possible.

Type Inference Accuracy. Following prior work [59], we evaluate type inference accuracy using precision, recall, and F1 score for local variables, function argument variables, and function return types.

7. Evaluation

Our evaluation consists of four parts: a comparative evaluation, an ablation study, a human evaluation, and a malware case study. In the comparative study, we evaluate OXIDIZER against existing C and Rust decompilation

techniques regarding conciseness and fidelity losses (Section 7.1). We then study the contribution of each component in OXIDIZER through an ablation study (Section 7.2). We also conduct a user study to measure if and how much OXIDIZER improves humans' performance in Rust reverse engineering tasks (Section 7.3). Finally, we include several case studies to highlight the advantages of OXIDIZER in more concrete scenarios (Section 7.4).

7.1. Comparative Evaluation

Datasets. We include 27 projects from the top 50 Rust repositories on GitHub (ranked by stars), excluding the Rust compiler repository, tutorials, non-Rust projects (e.g., those embedding Rust libraries), and projects that we could not build for non-trivial reasons. We also include the Rust reimplementation of GNU Coreutils, as GNU Coreutils is widely used in prior work on decompilation [6–8]. The complete list of selected Rust projects can be found in Appendix A. We compiled these projects on Ubuntu 22.04 LTS with several optimization levels (O0, O1, O2, O3, Os, and Oz) and two different versions of Rust compilers (nightly-2025-05-22 and nightly-2023-05-22[2]). Additionally, we disable link-time optimization (LTO) and LLVM inlining to minimize the effects of function inlining. With inlining enabled, source-level baselines become unreliable for certain metrics (e.g., matched strings and matched functions), because inlined functions no longer one-to-one correspond with source functions. We provide evaluation results on inlining-enabled binaries (nightly-2025-05-22, O3) in Appendix D to demonstrate OXIDIZER's effectiveness under different settings. All binaries are stripped.

Method. For conciseness and fidelity, we evaluate OXIDIZER using the metrics described in Section 6 and compare against existing decompilers including angr, Hex-Rays, Ghidra, Binary Ninja (with Pseudo C representation), and Binary Ninja (with Pseudo Rust representation). Although Ghidrust is another potential baseline, we exclude it from our experiments due to its high failure rate. In particular, when tested on the 30,151 functions that Ghidra successfully decompiled, Ghidrust failed to transpile 80% of them. We only include functions that are successfully decompiled by all decompilers.

Results. We present results for one configuration (nightly-2025-05-22 with O3). OXIDIZER similarly outperforms other decompilers under the other two representative configurations (2025–O0 and 2023–O3) which are presented in Appendix D. All decompilers successfully decompiled 12,472 functions in the 2025-O3 dataset. OXIDIZER outperforms all other decompilers on all metrics except for matched function calls (marginally behind Binary Ninja). This is primarily because unmatched function calls are mostly caused by function inlining, but OXIDIZER does not implement function outlining because it is out of scope.

2. We use nightly Rust compiler versions so that we can use the unstable option `-Z inline-llvm=false` to disable LLVM inlining.

A notable result is that the average number of variables produced by OXIDIZER (10.65) remains significantly higher than in the original source code (0.70). We investigated this gap and found that it is largely due to Rust's functional programming style, which encourages the use of chained expressions. As a result, Rust code often avoids introducing intermediate variables. However, none of the evaluated decompilers including OXIDIZER actively reconstruct such chained expressions. It is also interesting that Binary Ninja (Pseudo Rust) produces 21% fewer operators compared to Binary Ninja (Pseudo C). That is because Binary Ninja (Pseudo Rust) converts algorithmic memory access expressions to macro-like annotations, which is neither considered as a function call nor an operator.

OXIDIZER is also the only decompiler that can recover macro calls. It correctly recovers 9.8% of the macro calls from the source. OXIDIZER fails to recover the remaining targeted macros mainly because compiler optimizations introduce unexpected control flow and data flow, or because the macros are user-defined or standard macros not covered by OXIDIZER.

Table 2 presents the precision, recall, and F1 scores for type inference achieved by OXIDIZER and existing decompilers. The majority of variables in our datasets are of struct types (16,681 out of 53,113), where OXIDIZER achieves a precision of 8.08% and a recall of 29.89%, outperforming all other decompilers. Notably, OXIDIZER is the only decompiler capable of recovering enum types. We separate `Option<T>`, `Result<T, E>` from other enum types because of their prevalence among all enum types. As discussed in Section 6, all evaluated decompilers tend to produce a large number of variables that do not exist in the original source code. We consider any inferred type associated with such non-existent variables as a false positive, which contributes to the low precision across many type categories and decompilers.

7.2. Ablation Study

We perform a cascade analysis by disabling each of the following three components in OXIDIZER individually and examining how the removal of each component affects both its direct output and downstream stages.

Rust Standard Library Function Identification. Disabling Rust standard library function identification means OXIDIZER cannot use any function symbols. In this case, the macro collapsing component fails because Rust standard library function calls are part of macro expansion patterns. The number of extraneous calls increased from 2.79 to 3.52 because OXIDIZER cannot identify resource-release functions to remove. These results highlight the critical role of symbol information in enabling effective Rust decompilation.

FLIRT Signature Propagation. The direct impact of disabling this component is a drop in standard library function identification from 12% to 8%. The impact is more pronounced for resource-release function identification, which

TABLE 1: Evaluation results on conciseness and fidelity metrics for the nightly-2025-05-22-O3 dataset. 12,472 functions are evaluated across 157 binaries. We present average and median values per function for each metric. We also show percentage relative to the source code in parentheses for average values. The best average values per metric are highlighted in **bold**.

Metric	Source		OXIDIZER		angr		Hex-Rays		Ghidra		Binary Ninja (Pseudo C)		Binary Ninja (Pseudo Rust)	
	Avg.	Med.	Avg.	Med.	Avg.	Med.	Avg.	Med.	Avg.	Med.	Avg.	Med.	Avg.	Med.
CC	2.99	2	**3.20 (107.3%)**	1	3.90 (130.7%)	2	4.14 (138.8%)	2	4.07 (136.4%)	2	4.23 (141.7%)	2	4.23 (141.6%)	2
LoC	21.71	11	**41.06 (189.1%)**	21	49.58 (228.4%)	25	47.88 (220.6%)	23	61.21 (281.9%)	29	53.75 (247.6%)	25	56.07 (258.3%)	26
NofVar	0.70	0	**10.65 (1,530.3%)**	5	12.20 (1,752.8%)	6	13.74 (1,974.2%)	8	16.16 (2,321.7%)	7	15.39 (2,211.6%)	8	15.39 (2,211.6%)	8
NofOps	3.01	1	**21.39 (711.0%)**	9	31.27 (1,039.3%)	13	30.53 (1,014.8%)	11	31.55 (1,048.6%)	14	29.38 (976.4%)	14	23.09 (767.4%)	10
Gotos	0.00	0	**0.28**	0	0.51	0	0.79	0	0.74	0	0.50	0	0.50	0
Matched Strings	1.38	0	**0.62 (45.2%)**	0	0.10 (7.0%)	0	0.00 (0.3%)	0	0.39 (28.5%)	0	0.04 (3.1%)	0	0.04 (3.1%)	0
Matched Calls	7.78	4	2.53 (32.5%)	1	2.57 (33.0%)	1	2.88 (37.1%)	1	2.00 (25.7%)	1	**2.92 (37.6%)**	1	2.92 (37.6%)	1
Extraneous Calls	0.00	0	**2.79**	1	3.67	1	3.88	1	3.08	1	4.10	1	4.10	1
Matched Macros	0.67	0	**0.07 (9.8%)**	0	0.00 (0.0%)	0	0.00 (0.0%)	0	0.00 (0.0%)	0	0.00 (0.0%)	0	0.00 (0.0%)	0

TABLE 2: Type inference evaluation results across decompilers for the nightly-2025-05-22-O3 dataset. The precision, recall, and F1 score are reported per type category, and the best scores are highlighted in **bold**. The total number of types per category is also provided. For struct and enum types, OXIDIZER outperforms other decompilers on all scores.

Category	OXIDIZER			angr			Hex-Rays			Ghidra			Binary Ninja			Total
	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	
Overall	**4.57%**	**15.40%**	**7.05%**	0.84%	3.63%	1.36%	1.15%	5.20%	1.88%	0.79%	4.18%	1.32%	0.75%	4.27%	1.27%	53,113
Primitive	**1.87%**	30.34%	**3.53%**	1.07%	35.05%	2.07%	1.41%	**52.02%**	2.74%	1.01%	42.00%	1.97%	1.22%	41.57%	2.38%	4,771
Reference	**3.16%**	**2.40%**	**2.73%**	0.38%	1.05%	0.55%	0.45%	0.90%	0.60%	0.27%	0.86%	0.42%	0.19%	1.16%	0.33%	11,885
Array	0.00%	0.00%	0.00%	0.02%	0.79%	0.03%	**0.11%**	**11.81%**	**0.22%**	0.09%	6.30%	0.17%	0.00%	0.00%	0.00%	127
Struct	**8.08%**	**29.89%**	**12.72%**	0.00%	0.00%	0.00%	2.53%	0.27%	0.48%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	16,681
Option<T>	**3.28%**	**17.72%**	**5.53%**	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	2,049
Result<T, E>	**6.21%**	**18.67%**	**9.32%**	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	3,536
Other Enum	**35.25%**	**7.02%**	**11.71%**	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	2,180

drops from 23% to 3%, as most such functions are identified through propagated signatures. As a consequence, `# Extraneous Calls` increases from 2.79 to 3.34, as OXIDIZER can no longer remove calls to unrecognized resource-release functions. This also highlights the improvements of OXIDIZER's Rust symbol recovery over FLIRT.

Function Prototype Inference. In the nightly-2025-O3 dataset, 66% functions are user-defined functions. Disabling function prototype inference means OXIDIZER can no longer infer parameter and return types for user-defined functions and the unrecognized Rust standard library functions. With this component disabled, struct recovery recall significantly drops to around 6% and enum recovery recall drops to nearly zero. This is because without function prototype inference, constraint-based type inference can only infer struct and enum types from known Rust standard library functions. Furthermore, the number of recovered `match` and `if let` expressions drops by 77% and 85% respectively (from 3,138 to 721 and 3,926 to 579), as accurate function prototypes are essential for identifying enum discriminant checks and reconstructing pattern matching logic.

7.3. Human Evaluation

Method. To measure the impact of OXIDIZER on humans' performance in Rust reverse engineering tasks, we conduct an A/B test by dividing participants into two groups, OXIDIZER group and Hex-Rays group respectively. Each participant is assigned four reverse engineering challenges, each targeting a distinct capability: environment checking, file encryption, reverse shell generation, and command-line parsing. Participants self-reported their reverse engineering expertise level (expert, intermediate, or beginner) prior to the study.

Results. We collected three types of results from the human study: the scores participants achieved in the study, the time they spent to finish the study, and their perception of the decompilation quality.

Table 3 presents the average scores and completion times of participants across different expertise levels. The results indicate that participants generally achieved higher scores and completed the tasks more quickly when using OXIDIZER compared to Hex-Rays, which suggests that OXIDIZER produces decompiled output that is easier to understand.

In addition to objective task scores, user ratings further highlight a clear preference for OXIDIZER. Across all tasks, participants consistently rated OXIDIZER more favorably, with an overall average rating of 4.49 out of 5, compared to 2.61 for Hex-Rays. User comments consistently favored OXIDIZER's output for its readability and Rust-like structure, while Hex-Rays was often seen as more verbose and harder to follow. In summary, users not only performed better with OXIDIZER, but also perceived its decompilation output as significantly more understandable and helpful.

7.4. Case Study

Motivating Example. Compared to the three decompilation or reconstruction outputs shown in Listing 2, OXIDIZER produces the result closest to the original Rust source code (as Listing 1 shows).

First, OXIDIZER removes extraneous resource-release code, making the control flow much clearer. Second, OXI-

TABLE 3: Average task scores and completion times across self-reported reverse engineering expertise levels. The participant split between the OXIDIZER and Hex-Rays groups is shown for each expertise level.

Expertise	Participants	Split	OXIDIZER		Hex-Rays	
			Score (%)	Time (sec.)	Score (%)	Time (sec.)
All	37	18 / 19	**86.87**	**1,951**	67.94	2,449
Expert	5	3 / 2	**93.94**	**2,179**	81.82	3,368
Intermediate	22	10 / 12	**87.27**	**1,907**	70.45	2,195
Beginner	10	5 / 5	**81.82**	**1,901**	56.36	2,693

TABLE 4: Evaluation results on conciseness and the # `Gotos` metric for stripped malware samples. 1,180 functions are evaluated. OXIDIZER outperforms Hex-Rays on all conciseness metrics, and produces fewer gotos.

Metric	OXIDIZER		Hex-Rays	
	Avg.	Med.	Avg.	Med.
CC	**12.63**	7	20.25	10
LoC	**103.52**	57	149.47	77
NofVar	**23.80**	13	29.43	17
NofOps	**76.53**	36	96.77	45
Gotos	**1.99**	0	5.75	1

DIZER recovers Rust error handling constructs `match` and `if let`. These structures are essential for understanding control flow and error handling in Rust. In contrast, Hex-Rays decompile these into C-style control structures, losing the original high-level semantics, while ChatGPT's reconstruction wrongly presents error propagation operators, which do not align with the source.

Despite these advantages, OXIDIZER still has limitations. It fails to recover the types of function arguments `a1` and `a2` as `&[u8]`.

Malware Analysis. We conduct an evaluation on eight stripped Rust malware samples that we collected from [13]. The complete list of selected Rust projects can be found in Appendix B. Table 4 shows that OXIDIZER outperforms Hex-Rays on real-world samples in terms of conciseness. We do not include fidelity metrics since we do not have the source code of these malware samples to compare with.

We also conduct a case study to demonstrate how OXIDIZER and Hex-Rays perform differently on real-world malware samples. Listing 6 contains the mode selection code in Luna [60] as decompiled by OXIDIZER and Hex-Rays, respectively. OXIDIZER's Rust-specific simplifications result in decompiled code that is much easier to understand: (1) The extraneous code removal simplifications eliminate bounds checking code and memory deallocation code; (2) OXIDIZER recovers the expanded `println!` macro calls, with the string pieces and arguments. Listing 7 shows a comparison between OXIDIZER and Hex-Rays for a snippet from the `remove_snapshots` function, which lists virtual machines by executing a command. This function internally executes the ESXi command `vim-cmd vmsvc/getallvms`, and the result is propagated using Rust's `?` operator to handle errors concisely.

Based on the recovered types, OXIDIZER successfully identifies the error-propagation construct and simplifies the

(a) OXIDIZER

```
if std::path::Path::is_file(*(v14 as &i64), *((v14 +
    16) as &i64)) {
    println!("{} {} {} encrypting", v12, v14 - 24,
        v14);
    ...
}
```

(b) Hex-Rays

```
std::path::Path::is_file();
if ( v8 ) {
    if ( v1 <= v5 ) {
        core::panicking::panic_bounds_check();
    }
    if ( v1 <= v4 ) {
        core::panicking::panic_bounds_check();
    }
    *(_QWORD *)&v23 = v0;
    *((_QWORD *)&v23 + 1) = ::fmt;
    *(_QWORD *)&v24 = v2 - 24;
    *((_QWORD *)&v24 + 1) = ::fmt;
    v25 = (__int128 *)v2;
    v26 = ::fmt;
    *(_QWORD *)&v27 = &off_74480;
    *((_QWORD *)&v27 + 1) = 4LL;
    *(_QWORD *)&v28 = 0LL;
    v29 = &v23;
    v30 = 3LL;
    std::io::stdio::_print();
    ...
}
```

Listing 6: The mode selection logic in Luna as decompiled by OXIDIZER and Hex-Rays.

decompilation into idiomatic Rust syntax using the `?` operator. This makes the high-level semantics of the original code much easier to understand. By contrast, the Hex-Rays output shows low-level pointer comparisons and conditional branches, making it harder to identify the intent of error propagation.

With these recovered Rust high-level abstractions, malware analysts can quickly focus on the normal execution flow and reason about the core functionality without being distracted by the low-level noise introduced by compilation.

LLM4Decompile. We evaluate LLM4Decompile [61] (6.7B-v2) by feeding it Ghidra's decompilation output for 1,562 functions from the Rust coreutils binaries. Of these, 1,318 functions (84.4%) produced *degenerate output*: the model either repeatedly generated the same tokens or was truncated before completing the decompilation. For the remaining 244 functions, we randomly sampled 30 functions for manual inspection and found 28 of them (93.3%) suffered from hallucination, including fabricated struct types, missing code, incorrect constants, invented control-flow logic, and broken syntax. These results indicate that current fine-tuned LLM-based decompilers remain unreliable for practical Rust decompilation. Beyond reliability, LLM4Decompile produces only textual C output and cannot recover Rust-specific abstractions such as enums, pattern matching, or macros.

8. Limitations

In this section, we discuss some of the limitations in OXIDIZER and areas for future work in Rust decompilation.

(a) OXIDIZER

```
fn compat_core::esxi::vm::remove_snapshots() ->
↪  Result<(), Error> {
    ...
    v25 = compat_core::esxi::utils::esxi_run_comman
    ↪  d_with_output("vim-cmd vmsvc/getallvms")?;
    ...
}
```

(b) Hex-Rays

```
__int64
↪  compat_core::esxi::vm::remove_snapshots(__int64
↪  *a1)
{
    ...
    compat_core::esxi::utils::esxi_run_command_with
    ↪  _output(&v46, aVimCmdVmsvcGet, 23LL);
    v2 = v48;
    if ( v46 == (char **)((char *)&dword_0 + 1) ) {
        *a1 = v47;
        a1[1] = v2;
        return v1;
    }
    ...
}
```

Listing 7: The VM listing logic in BlackCat (Sphynx) as decompiled by OXIDIZER and Hex-Rays. In OXIDIZER's decompilation (Listing 7a), we can clearly see that the ESXi command `vim-cmd vmsvc/getallvms` is executed, and the result is propagated using Rust's `?` operator to handle errors concisely.

Generalizability. OXIDIZER relies on pattern-matching heuristics tailored to specific compiler behaviors, which may break when new `rustc` versions change code generation patterns, enum layouts, or ABI conventions. Adapting to a new compiler version requires regenerating version-specific type databases and potentially updating analysis rules. Additionally, Oxidizer targets a fixed set of high-level constructs (structs, enums, and macros) and does not recover other Rust abstractions such as traits, generics, or closures.

Compiler Optimization. A noticeable consequence of Rust compiler optimization is the presence of gotos in Rust binaries. Compiler optimization also hinders OXIDIZER from recovering Rust high-level abstractions during macro recovery, string recovery, and so on, by causing unexpected control flow and data flow in decompilation. Previous researchers have found that compiler optimization is the cause of extraneous gotos in C decompilation [8], and they revert compiler optimization before structuring to avoid unexpected schemas that cause extraneous gotos. The same idea of reverting compiler optimization may also apply to Rust decompilation, but we leave that to future work.

Lossy Compilation. Decompilation is challenging due to the loss of information inherent in the compilation process, notably in high-level types [62]. For example, a function that returns a struct may look the same as one that returns an enum with two variants in decompilation, if the two variants are the same size as the struct. It may be hard, or even impossible, for decompilers to distinguish these two cases deterministically. Recent researchers have used machine learning and LLMs to overcome the indeterminacy of decompilation [14–16, 59].

Obfuscation. Similar to previous work [6–8], OXIDIZER is not resilient to anti-static analysis techniques, including packing and other code obfuscation techniques. As a static decompiler, OXIDIZER also cannot handle self-modifying code (SMC) or JIT-compiled code, which may appear in Rust malware. Instead, we rely on previous work on general deobfuscation, some of which operate on the control-flow graph level [63] or binary level [64], and would be agnostic to Rust.

Recompilation. OXIDIZER does not generate recompilable Rust code. State-of-the-art C decompilers, including angr, Hex-Rays, Binary Ninja, and Ghidra, also do not guarantee recompilability. Previous research has explored reassembly [65, 66] and recompilable IRs [67], whereas to the best of our knowledge, there is no research work on recompilable decompilation.

Language Applicability. Although challenges in decompiling Rust binaries are similar to those in other high-level languages, addressing new languages, such as Swift, requires a different set of language-specific research. We inherently designed OXIDIZER to work on Rust; as a result, this limits the applicability of our techniques to other languages. However, we maintain that OXIDIZER will help broaden high-level decompilation research beyond the C/C++ family and improve program understanding for languages increasingly used in both benign and malicious software.

9. Related Work

Rust Support in C Decompilers. To the best of our knowledge, there is no publicly available specialized Rust decompiler. Analysts often use state-of-the-art C decompilers including angr [18], Hex-Rays [10], Ghidra [11], and Binary Ninja [21] to analyze Rust binaries. Binary Ninja introduced Pseudo Rust representation since 4.2 [29], similar to OXIDIZER's Rust pseudocode generator component, which translates the structured AST into human-readable Rust pseudocode. Ghidrust [30] is a Rust binary analysis extension for Ghidra, which supports emitting Rust pseudocode by parsing decompiled C code. Ghidrust also supports identifying Rust binaries by searching for the error messages used in Rust standard library functions. Aside from Rust support in C decompilers, there is also a transpiler C2Rust [68] that migrates C99-compliant compilable C code to Rust.

LLM-assisted Decompilation. Large language models (LLMs) have demonstrated remarkable performance in decompilation quality improvement. Notably, DeGPT [14] represents one of the earliest efforts in this direction, employing off-the-shelf LLMs as collaborative agents to refine decompiler outputs and improve readability. Other works, such as ReSym [16], employ LLMs for recovering variable names and types, while TypeForge [17] focuses specifically on reconstructing composite variable types. The most recent work, FidelityGPT [15], tackles fidelity distortion in decompiled outputs by leveraging distortion-aware prompting,

retrieval-augmented generation, and variable dependency analysis to detect and correct semantic inconsistencies between decompiled code and its source, thereby enhancing its readability and accuracy. Even though they are not directly comparable with OXIDIZER as they either aim to improve decompilation readability or improve one component in the decompilation pipeline, these LLM-based techniques could potentially be adapted to improve Rust decompilation.

Binary Decompilation. Since Cifuentes et al. laid the academic foundation [58, 69] for modern decompilers, many efforts have been made to improve the performance of decompilers. The scientific challenges in binary decompilation mainly include control-flow structuring, type recovery and variable name recovery. Phoenix [6] uses *semantics-preservation* and *iterative refinement* to reduce the number of gotos. DREAM [7] and rev.ng [53] further introduce algorithms to eliminate gotos. SAILR [8] reduces gotos by inverting compiler-aware transformations to maintain high structure similarity to the source code. To achieve type inference in binary decompilation, previous works have used program analysis techniques (TIE [70], Retypd [52], Osprey [71]) and machine learning techniques (DIRTY [72], TYGR [59]). High-precision variable name prediction is especially useful for making decompilation more understandable by humans. Many recent works use neural models for variable name prediction (DIRE [73], DIRTY [72], VarBERT [74], ReSym [16]). Unfortunately, these techniques target C decompilations and improve decompilation quality in general, but do not address the challenges in decompiling Rust binaries.

10. Conclusion

Decompilation plays a critical role in binary analysis, yet existing decompilers are designed primarily for C/C++ binaries and fall short when applied to Rust binaries directly. This is due to the fundamental differences in language design and semantics. In this work, we present the first comprehensive study of the challenges in Rust decompilation, identifying the fidelity issues and the underlying causes. Based on these insights, we propose a set of targeted techniques tailored for concise and high-fidelity Rust decompilation, presenting our prototype OXIDIZER. We conduct extensive evaluation to demonstrate the efficacy of OXIDIZER, making a significant step towards Rust binary analysis.

Acknowledgements

This paper has received funding from the Air Force Office of Scientific Research No. FA9550-24-1-0204; the Advanced Research Projects Agency for Health (ARPA-H) No. SP4701-23-C-0074; the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) No. N66001-20-C-4020; and the National Science Foundation No. 2232915 and 2146568.

Ethics Considerations

This research on Rust decompilation was conducted with careful attention to ethical considerations. The goal of the research was to facilitate security analyses, such as malware analysis and binary auditing, not to facilitate software piracy, unauthorized access, or the circumvention of security mechanisms. The human study in this research was exempted by our Institutional Review Board (IRB) at Arizona State University. However, we followed the same ethics and privacy requirements that an IRB would normally enforce.

LLM Usage Considerations

LLMs were used for editorial purposes in this paper, and all outputs were inspected by the authors to ensure accuracy and originality. The methodology of this paper does not integrate LLMs or any ideas generated by LLMs. No LLMs were trained or fine-tuned in this process.

References

[1] S. Klabnik and C. Nichols, *The Rust programming language*. No Starch Press, 2023.

[2] Airborne Engineering Limited, "blethrs," https://github.com/airborneengineering/blethrs.

[3] H. Li, L. Guo, Y. Yang, S. Wang, and M. Xu, "An empirical study of {Rust-for-Linux}: The success, dissatisfaction, and compromise," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, 2024, pp. 425–443.

[4] J. Corbet, "Rust for embedded linux kernels," https://lwn.net/Articles/970216/, Apr. 2024.

[5] H. Evans, "5 malware variants you should know," https://www.reliaquest.com/blog/5-malware-variants-you-should-know/, Aug. 2024.

[6] D. Brumley, J. Lee, E. J. Schwartz, and M. Woo, "Native x86 decompilation using Semantics-Preserving structural analysis and iterative Control-Flow structuring," in *22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX Association, Aug. 2013, pp. 353–368. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/schwartz

[7] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith, "No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformations," in *NDSS*. Citeseer, 2015.

[8] Z. L. Basque, A. P. Bajaj, W. Gibbs, J. O'Kain, D. Miao, T. Bao, A. Doupé, Y. Shoshitaishvili, and R. Wang, "Ahoy SAILR! there is no need to DREAM of c: A Compiler-Aware structuring algorithm for binary decompilation," in *33rd USENIX Security Symposium (USENIX Security*

24). Philadelphia, PA: USENIX Association, Aug. 2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/basque

[9] M. Botacin, "What do malware analysts want from academia? a survey on the state-of-the-practice to guide research developments," in *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses*, 2024, pp. 77–96.

[10] Hex-Rays, "Disassemble, decompile and debug with ida," 2025. [Online]. Available: https://hex-rays.com/

[11] NSA, "Ghidra," 2025. [Online]. Available: https://ghidra-sre.org/

[12] N. Fishbein and J. A. Guerrero-Saade, "Project 0xa11c deoxidizing the rust malware ecosystem," https://www.youtube.com/watch?v=3WsTYSUz-UQ, Oct. 2024.

[13] C. Xiao, "Rust malware gallery," 2024. [Online]. Available: https://github.com/cxiao/rust-malware-gallery

[14] P. Hu, R. Liang, and K. Chen, "Degpt: Optimizing decompiler output with llm," in *Proceedings 2024 Network and Distributed System Security Symposium*, 2024.

[15] Z. Zhou, X. Li, R. Feng, Y. Zhang, Y. Li, W. Feng, Y. Wang, and Y. Li, "Fidelitygpt: Correcting decompilation distortions with retrieval augmented generation," in *Proceedings 2026 Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2026.

[16] D. Xie, Z. Zhang, N. Jiang, X. Xu, L. Tan, and X. Zhang, "Resym: Harnessing llms to recover variable and data structure symbols from stripped binaries," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 4554–4568.

[17] Y. Wang, R. Liang, Y. Li, P. Hu, K. Chen, and B. Zhang, "Typeforge: Synthesizing and selecting best-fit composite data types for stripped binaries," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2025, pp. 2847–2864.

[18] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 138–157.

[19] uutils, "uutils coreutils," 2025. [Online]. Available: https://github.com/uutils/coreutils

[20] Neyrian, "Fakecrypt: Easy to use ransomware-like tool for linux or windows, developed in rust," GitHub repository, 2025.

[21] Vector35, "Binary ninja," 2025. [Online]. Available: https://binary.ninja/

[22] OpenAI, "Chatgpt (gpt-5)," https://chat.openai.com/, 2025.

[23] Z. Xu, S. Jain, and M. S. Kankanhalli, "Hallucination is inevitable: An innate limitation of large language models," arXiv preprint arXiv:2401.11817, 2024. [Online]. Available: https://arxiv.org/abs/2401.11817

[24] Microsoft Security Response Center, "Why Rust for safe systems programming," https://msrc.microsoft.com/blog/2019/07/why-rust-for-safe-systems-programming/, 2019.

[25] Microsoft Threat Intelligence, "Hive ransomware gets upgrades in Rust," https://www.microsoft.com/en-us/security/blog/2022/07/05/hive-ransomware-gets-upgrades-in-rust/, 2022.

[26] Intezer, "WildCard: The APT Behind SysJoker Targets Critical Sectors in Israel," https://intezer.com/blog/wildcard-evolution-of-sysjoker-cyber-threat/, 2023.

[27] Rust Project Developers, "The rustc-dev-guide," https://rustc-dev-guide.rust-lang.org/overview.html, 2024, accessed: 2026-03-24.

[28] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith, "Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 158–177.

[29] Binary Ninja Team, "Binary Ninja 4.2 "Frogstar" Released," Nov. 2024. [Online]. Available: https://binary.ninja/2024/11/20/4.2-frogstar.html

[30] DMaroo, "Ghidrust," 2024, available at: https://github.com/DMaroo/GhidRust (Accessed: 2024-11-03).

[31] L. Dramko, J. Lacomis, E. J. Schwartz, B. Vasilescu, and C. L. Goues, "A taxonomy of c decompiler fidelity issues," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 379–396. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/dramko

[32] M. F. Oberhumer, L. Molnár, and J. F. Reiser, "Upx: The ultimate packer for executables," 2025. [Online]. Available: https://github.com/upx/upx

[33] O. Technologies, "Themida – advanced windows software protection system," 2025. [Online]. Available: https://www.themida.com/

[34] V. Software, "Vmprotect – software protection utility," 2025. [Online]. Available: https://vmpsoft.com/vmprotect/

[35] G. Iaculo, "rustfuscator," GitHub repository, 2025. [Online]. Available: https://github.com/gianiac/rustfuscator

[36] P. Dronavalli, "Rust-obfuscator," GitHub repository, 2025. [Online]. Available: https://github.com/dronavallipranav/rust-obfuscator

[37] frank2, "goldberg," GitHub repository, 2025. [Online]. Available: https://github.com/frank2/goldberg

[38] joaovarelas, "Obfuscator-llvm-16.0," GitHub repository, 2025. [Online]. Available: https://github.com/joaovarelas/Obfuscator-LLVM-16.0

[39] 0xlane, "ollvm-rust," GitHub repository, 2025. [Online]. Available: https://github.com/0xlane/ollvm-rust

[40] The Rust Language Project, *The Rust Reference — Enumerations*, 2025. [Online]. Available: https://doc.rust-lang.org/reference/items/enumerations.html

[41] F. S. Foundation, *Link Options — GNU Compiler Collection (GCC) Documentation*, 2025. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html

[42] T. L. Project, *Clang — The Clang C, C++, and Objective-C Compiler / Command Guide*, 2025. [Online]. Available: https://clang.llvm.org/docs/CommandGuide/clang.html

[43] The Rust Project, *The Rust Reference — Linkage*, 2025. [Online]. Available: https://doc.rust-lang.org/reference/linkage.html

[44] J. Thagen, "min-sized-rust: How to minimize rust binary size," GitHub repository, 2025. [Online]. Available: https://github.com/johnthagen/min-sized-rust

[45] cppreference.com contributors. (2025) malloc – c reference. [Online]. Available: https://en.cppreference.com/w/c/memory/malloc

[46] Steve Klabnik and Carol Nichols and The Rust Project, *The Rust Programming Language*. No Starch Press, 2025.

[47] International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), *Programming languages — C*, 1st ed. ISO/IEC, 1990.

[48] The Rust Project, "Strings – rust by example / the big book of rust interop," https://doc.rust-lang.org/rust-by-example/std/str.html and https://nrc.github.io/big-book-ffi/reference/strings.html, 2025.

[49] cppreference.com contributors. (2025) Struct declaration — c reference. [Online]. Available: https://en.cppreference.com/w/c/language/struct.html

[50] The Rust Language Project, *The Rust Reference — Type Layout*, 2025. [Online]. Available: https://doc.rust-lang.org/reference/type-layout.html

[51] Hex-Rays, *Fast Library Identification and Recognition Technology (FLIRT)*, n.d. [Online]. Available: https://docs.hex-rays.com/user-guide/signatures/flirt

[52] M. Noonan, A. Loginov, and D. Cok, "Polymorphic type inference for machine code," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 27–41.

[53] A. Gussoni, A. Di Federico, P. Fezzardi, and G. Agosta, "A comb for decompiled c code," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 637–651. [Online]. Available: https://doi.org/10.1145/3320269.3384766

[54] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[55] I. Hosseini and B. Dolan-Gavitt, "Beyond the C: Retargetable decompilation using neural machine translation," in *Workshop on Binary Analysis Research (BAR)*, 2022.

[56] S. Enders, E.-M. C. Behner, N. Bergmann, M. Rybalka, E. Padilla, E. X. Hui, H. Low, and N. Sim, "dewolf: Improving decompilation by leveraging user surveys," in *Proceedings 2023 Workshop on Binary Analysis Research*, ser. BAR 2023. Internet Society, 2023. [Online]. Available: http://dx.doi.org/10.14722/bar.2023.23001

[57] S. Bhatia and J. Malhotra, "A survey on impact of lines of code on software complexity," in *2014 International Conference on Advances in Engineering & Technology Research (ICAETR-2014)*. IEEE, 2014, pp. 1–4.

[58] C. Cifuentes, *Reverse compilation techniques*. Queensland University of Technology, Brisbane, 1994.

[59] C. Zhu, Z. Li, A. Xue, A. P. Bajaj, W. Gibbs, Y. Liu, R. Alur, T. Bao, H. Dai, A. Doupé *et al.*, "TYGR: Type inference on stripped binaries using graph neural networks," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 4283–4300.

[60] Kaspersky, "Luna in Rust: new ransomware group emerges using cross-platform programming language," https://www.kaspersky.com/about/press-releases/luna-in-rust-new-ransomware-group-emerges-using-cross-platform-programming-language, 2022.

[61] H. Tan, Q. Luo, J. Li, and Y. Zhang, "Llm4decompile: Decompiling binary code with large language models," *arXiv preprint arXiv:2403.05286*, 2024.

[62] W. K. Wong, H. Wang, Z. Li, Z. Liu, S. Wang, Q. Tang, S. Nie, and S. Wu, "Refining decompiled c code with large language models," The Hong Kong University of Science and Technology, Tech. Rep., 2023. [Online]. Available: https://arxiv.org/pdf/2310.06530v2

[63] OALabs, "Control-flow deobfuscation," 2022. [Online]. Available: https://research.openanalysis.net/angr/symbolic%20execution/deobfuscation/research/2022/03/26/angr_notes.html

[64] S. Li, C. Jia, P. Qiu, Q. Chen, J. Ming, and D. Gao, "Chosen-instruction attack against commercial code virtualization obfuscators," in *Proceedings 2022 Network and Distributed System Security Symposium (NDSS)*, 2022.

[65] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making reassembly great again," in *24th Annual Network and Distributed System Security Symposium (NDSS 2017)*. The Internet Society, 2017.

[66] H. Kim, S. Kim, J. Lee, K. Jee, and S. K. Cha, "Reassembly is hard: A reflection on challenges and strategies," in *32nd USENIX Security Symposium (USENIX Security '23)*. USENIX Association, 2023, p. 1469–1486. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/kim-hyungseok

[67] A. Altinay, J. Nash, T. Kroes, P. Rajasekaran, D. Zhou, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, C. Giuffrida, H. Bos, and M. Franz, "Binrec: Dynamic binary lifting and recompilation," in *Proceedings of the 15th European Conference on Computer Systems (EuroSys '20)*, 2020.

[68] "C2rust demonstration," 2024, available at: https://c2rust.com/ (Accessed: 2024-11-03).

[69] C. Cifuentes and K. J. Gough, "Decompilation of binary programs," *Software: Practice and Experience*, vol. 25, no. 7, pp. 811–829, 1995.

[70] J. Lee, T. Avgerinos, and D. Brumley, "Tie: Principled reverse engineering of types in binary programs," in

NDSS, 2011.
[71] Z. Zhang, Y. Ye, W. You, G. Tao, W.-c. Lee, Y. Kwon, Y. Aafer, and X. Zhang, "Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 813–832.
[72] Q. Chen, J. Lacomis, E. J. Schwartz, C. Le Goues, G. Neubig, and B. Vasilescu, "Augmenting decompiler output with learned variable names and types," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 4327–4343.
[73] J. Lacomis, P. Yin, E. Schwartz, M. Allamanis, C. Le Goues, G. Neubig, and B. Vasilescu, "Dire: A neural approach to decompiled identifier renaming," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 628–639.
[74] K. K. Pal, A. P. Bajaj, P. Banerjee, A. Dutcher, M. Nakamura, Z. L. Basque, H. Gupta, S. A. Sawant, U. Anantheswaran, Y. Shoshitaishvili *et al.*, "" len or index or count, anything but v1": Predicting variable names in decompilation output with transfer learning," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 4069–4087.

Appendix A. Selected Popular Rust Projects for Evaluation

The 28 selected popular Rust projects we used for evaluation includes: rustdesk, uv, sway, alacritty, fuel-core, ripgrep, bat, meilisearch, starship, vaultwarden, typst, fuels-rs, ruff, helix, fd, lapce, nushell, swc, fish-shell, linera-protocol, sniffnet, influxdb, firecracker, zoxide, surrealdb, turborepo, just, coreutils.

Appendix B. Selected Real-world Malware Samples

The 8 selected real-world Rust malware samples we used for the case study evaluation are sourced from the Rust Malware Gallery [13]. They include: Luna Ransomware (`1cbbf1...ab51`), Realst Stealer (`2af0e2...13e2`), CosmicRust (`3315e5...29a`), Convuster (`947ae8...66a`), RustBucket (`9ca914...c747`), RansomExx2 (`a7ea1e...8c5c`), BlackCat Sphynx (`c0e70e...38cc`), and RustBucket (`de81e5...d500`).

Appendix C. Human Study Details

C.1. A Sample Recruitment Email

Hi,

We're conducting a human study on OXIDIZER, a research prototype Rust decompiler developed by the SEFCOM lab at Arizona State University. The goal is to evaluate whether OXIDIZER can assist with reverse engineering Rust programs.

We're looking for 30 participants with programming experience (Rust experience not required). The study will be conducted remotely, and upon successful completion, you'll receive a $50 Amazon gift card as reward.

The study is available at [REDACTED]. For more questions, feel free to email us at yiboliu@asu.edu.

Thank you!

C.2. An Example Task and Free-Text Question

Please read the provided code snippet and answer the following one question. You should answer "I don't know" if you still don't know the answer after your best effort.

Link: [A link to a webpage displaying the decompilation corresponding to participant's group]

Q1. Describe the functionality of this function with at most three sentences.

C.3. An Example Multi-choice Question

Q1. What will happen if this function fails to save the resulting file?
(a) It saves the file content to a temporary file instead
(b) It logs an error message and gracefully returns
(c) It panics immediately without any message
(d) It returns without logging
(e) I don't know

C.4. Example User Perception Questions

Q1. You can now compare the decompiled code you received with the output from another decompiler. Please evaluate and rate the quality of each version:

Link: [A link to a webpage displaying both decompilation outputs]

Snippet A: ☆ ☆ ☆ ☆ ☆
Snippet B: ☆ ☆ ☆ ☆ ☆

Q2. Do you have any additional comments for the quality of the decompiled code?

Appendix D.
Additional Evaluation Results for 2025-O0, 2023-O3, and 2025-O3-inline

TABLE 5: Evaluation results on the nightly-2025-05-22-O0 dataset. 12,549 functions are evaluated across 157 binaries. Ghidra's `CC`, `# Matched Calls`, and `# Extraneous Calls` metrics are abnormally lower compared to other decompilers, because Ghidra frequently fails to resolve indirect jumps, and therefore leads to missing code in the function.

Metric	Source		OXIDIZER		angr		Hex-Rays		Ghidra		Binary Ninja (Pseudo C)		Binary Ninja (Pseudo Rust)	
	Avg.	Med.	Avg.	Med.	Avg.	Med.	Avg.	Med.	Avg.	Med.	Avg.	Med.	Avg.	Med.
CC	2.63	1	3.73 (141.9%)	1	4.38 (166.4%)	1	3.62 (137.5%)	1	**3.51 (133.6%)**	2	4.72 (179.4%)	1	4.71 (179.2%)	1
LoC	17.70	8	**50.92 (287.6%)**	23	62.05 (350.5%)	26	84.40 (476.8%)	21	69.38 (391.9%)	23	60.81 (343.5%)	24	64.06 (361.9%)	26
NofVar	1.70	0	**18.42 (1,084.3%)**	8	21.24 (1,250.4%)	8	37.06 (2,181.8%)	11	23.99 (1,412.2%)	7	24.51 (1,442.9%)	10	24.51 (1,442.9%)	10
NofOps	2.62	0	**17.54 (669.0%)**	7	27.31 (1,041.5%)	9	21.82 (832.1%)	6	36.29 (1,383.7%)	11	24.97 (952.0%)	10	22.53 (859.2%)	7
Gotos	0.00	0	**0.14**	0	0.35	0	0.34	0	0.37	0	0.25	0	0.25	0
Matched Strings	1.08	0	**0.55 (51.1%)**	0	0.06 (5.7%)	0	0.01 (0.6%)	0	0.24 (22.5%)	0	0.04 (3.7%)	0	0.04 (3.6%)	0
Matched Calls	6.49	3	4.32 (66.6%)	2	4.40 (67.8%)	2	4.64 (71.5%)	2	1.87 (28.9%)	1	**4.75 (73.1%)**	2	4.75 (73.1%)	2
Extraneous Calls	0.00	0	3.29	1	4.53	1	4.90	1	**1.55**	0	5.05	1	5.05	1
Matched Macros	0.52	0	**0.05 (10.2%)**	0	0.00 (0.0%)	0	0.00 (0.0%)	0	0.00 (0.0%)	0	0.00 (0.0%)	0	0.00 (0.0%)	0

TABLE 6: Type inference evaluation results across decompilers for the nightly-2025-05-22-O0 dataset.

Category	OXIDIZER			angr			Hex-Rays			Ghidra			Binary Ninja			Total
	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	
Overall	**4.61%**	**23.20%**	**7.69%**	2.00%	11.69%	3.42%	1.51%	14.48%	2.73%	1.57%	10.16%	2.73%	1.94%	12.80%	3.37%	52,915
Struct	**5.67%**	**36.30%**	**9.81%**	0.00%	0.00%	0.00%	2.12%	0.12%	0.22%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	18,894
Option<T>	**3.35%**	**21.91%**	**5.81%**	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	2,173
Result<T, E>	**5.96%**	**16.60%**	**8.77%**	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	6,345

TABLE 7: Evaluation results on the nightly-2023-05-22-O3 dataset. 8,609 functions are evaluated across 130 binaries.

Metric	Source		OXIDIZER		angr		Hex-Rays		Ghidra		Binary Ninja (Pseudo C)		Binary Ninja (Pseudo Rust)	
	Avg.	Med.	Avg.	Med.	Avg.	Med.	Avg.	Med.	Avg.	Med.	Avg.	Med.	Avg.	Med.
CC	2.81	1	**2.77 (98.6%)**	1	3.01 (106.9%)	1	3.50 (124.5%)	1	3.62 (128.7%)	1	3.79 (134.6%)	1	3.79 (134.7%)	1
LoC	19.04	9	**39.46 (207.3%)**	19	45.54 (239.2%)	21	45.66 (239.9%)	21	61.78 (324.5%)	26	51.78 (272.0%)	22	49.08 (257.8%)	22
NofVar	0.72	0	**10.11 (1,406.8%)**	5	11.27 (1,568.0%)	5	13.77 (1,916.2%)	8	17.45 (2,427.9%)	7	14.66 (2,040.4%)	8	14.66 (2,040.3%)	8
NofOps	2.43	1	**20.59 (846.0%)**	8	29.13 (1,196.8%)	11	27.95 (1,148.3%)	9	28.34 (1,164.5%)	11	28.16 (1,156.9%)	12	21.88 (899.0%)	8
Gotos	0.00	0	**0.25**	0	0.35	0	0.69	0	0.59	0	0.46	0	0.46	0
Matched Strings	1.52	0	**0.81 (53.5%)**	0	0.05 (3.4%)	0	0.00 (0.0%)	0	0.40 (26.6%)	0	0.02 (1.2%)	0	0.02 (1.2%)	0
Matched Calls	7.45	3	3.13 (42.0%)	1	3.19 (42.7%)	1	3.43 (46.0%)	1	2.41 (32.3%)	1	3.46 (46.5%)	1	**3.46 (46.5%)**	1
Extraneous Calls	0.00	0	**3.16**	1	4.00	1	4.40	1	3.54	1	4.65	1	4.65	1
Matched Macros	0.57	0	**0.10 (17.2%)**	0	0.00 (0.0%)	0	0.00 (0.0%)	0	0.00 (0.0%)	0	0.00 (0.0%)	0	0.00 (0.0%)	0

TABLE 8: Type inference evaluation results across decompilers for the nightly-2023-05-22-O3 dataset.

Category	OXIDIZER			angr			Hex-Rays			Ghidra			Binary Ninja			Total
	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	
Overall	**5.68%**	**22.17%**	**9.05%**	1.04%	4.63%	1.69%	1.10%	5.75%	1.85%	0.82%	5.17%	1.42%	0.80%	4.37%	1.35%	28,701
Struct	**10.33%**	**33.42%**	**15.78%**	0.00%	0.00%	0.00%	1.23%	0.15%	0.27%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	12,843
Option<T>	**3.03%**	**15.03%**	**5.04%**	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	1,271
Result<T, E>	**8.60%**	**21.30%**	**12.25%**	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	2,873

TABLE 9: Evaluation results on the nightly-2025-05-22-O3-inline dataset. 12,349 functions are evaluated across 157 binaries.

Metric	Source		OXIDIZER		angr		Hex-Rays		Ghidra		Binary Ninja (Pseudo C)		Binary Ninja (Pseudo Rust)	
	Avg.	Med.	Avg.	Med.	Avg.	Med.	Avg.	Med.	Avg.	Med.	Avg.	Med.	Avg.	Med.
CC	3.27	2	**13.19 (402.7%)**	6	17.99 (549.3%)	9	22.11 (675.3%)	10	21.96 (670.5%)	9	20.96 (640.1%)	9	20.93 (639.3%)	9
LoC	26.06	14	**131.24 (503.6%)**	67	169.34 (649.8%)	86	187.48 (719.4%)	89	216.50 (830.8%)	106	208.39 (799.7%)	99	225.26 (864.4%)	107
NofVar	0.77	0	**33.12 (4,289.3%)**	17	37.15 (4,812.2%)	20	40.04 (5,185.9%)	24	35.57 (4,606.7%)	20	48.03 (6,221.5%)	25	48.04 (6,221.9%)	25
NofOps	3.57	1	**80.16 (2,244.6%)**	40	108.48 (3,037.8%)	54	140.71 (3,940.4%)	57	175.71 (4,920.4%)	73	116.54 (3,263.5%)	57	96.66 (2,706.7%)	46
Gotos	0.00	0	**2.01**	0	3.23	0	5.89	1	5.24	1	3.60	0	3.59	0
Matched Strings	1.23	0	**0.54 (44.0%)**	0	0.06 (5.3%)	0	0.02 (2.0%)	0	0.29 (23.9%)	0	0.08 (6.5%)	0	0.08 (6.5%)	0
Matched Calls	9.51	5	1.58 (16.6%)	1	1.58 (16.6%)	1	1.86 (19.6%)	1	0.74 (7.8%)	0	**1.87 (19.7%)**	1	1.87 (19.7%)	1
Extraneous Calls	0.00	0	7.63	3	10.77	5	11.99	5	**2.96**	1	13.04	6	13.05	6
Matched Macros	0.78	0	**0.11 (14.0%)**	0	0.00 (0.0%)	0	0.00 (0.0%)	0	0.00 (0.0%)	0	0.00 (0.0%)	0	0.00 (0.0%)	0

TABLE 10: Type inference evaluation results across decompilers for the nightly-2025-05-22-O3-inline dataset.

Category	OXIDIZER			angr			Hex-Rays			Ghidra			Binary Ninja			Total
	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	
Overall	**1.42%**	**14.81%**	**2.60%**	0.25%	2.94%	0.46%	0.43%	5.43%	0.79%	0.32%	3.64%	0.59%	0.25%	3.74%	0.47%	43,288
Struct	**2.08%**	**21.19%**	**3.79%**	0.00%	0.00%	0.00%	0.28%	0.34%	0.31%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	17,458
Option<T>	**1.46%**	**14.23%**	**2.66%**	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	2,171
Result<T, E>	**3.18%**	**19.92%**	**5.49%**	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	4,232

Appendix E.
Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

E.1. Summary

This paper introduces Oxidizer, a novel decompiler specifically designed to handle compiled Rust binaries by recovering high-level Rust abstractions such as enums and macros. The authors systematically identify why traditional C/C++ decompilers fail on Rust executables and evaluate Oxidizer using both quantitative metrics and a human-subject study, showing impressive improvements over existing state-of-the-art C decompilers.

E.2. Scientific Contributions

- Creates a New Tool to Enable Future Science.
- Provides a Valuable Step Forward in an Established Field.

E.3. Reasons for Acceptance

1) This paper provides a valuable step forward in an established field. The difficulty of analyzing Rust binaries with existing C/C++ decompilers is a recognized challenge. By systematically collecting and characterizing these failures, and providing a targeted and well-engineered solution, the authors demonstrate clear practical improvements in readability.
2) Furthermore, the paper creates a new tool to enable future science. Oxidizer represents the first open-source, dedicated decompiler for Rust. By making this tool available and validating its real-world effectiveness through a rare and highly appreciated human-subject study, the authors have laid a strong foundation for future research in Rust binary analysis.

E.4. Noteworthy Concerns

1) The proposed system relies heavily on specific pattern-matching and heuristics, which raises concerns about how brittle the tool might be against new compiler versions, aggressive optimizations, or future language features. The paper lacks a deeper discussion regarding a more principled, generalizable design approach to address the ”test-of-time” survival of these techniques.