# rustc++: Facilitating Advanced Analysis of Rust Code

Antonis Louka
louka.antonis@ucy.ac.cy
University of Cyprus
Nicosia, Cyprus

Georgios Portokalidis
georgios.portokalidis@imdea.org
IMDEA Software Institute
Madrid, Spain

Elias Athanasopoulos
athanasopoulos.elias@ucy.ac.cy
University of Cyprus
Nicosia, Cyprus

## ABSTRACT

Rust is a popular programming language with strong memory-safety guarantees, achieved through its ownership and borrowing model. Rust allows a programmer to escape memory safety using explicitly marked unsafe code in order to facilitate integration with existing C/C++ code. Such safe code is not immune to typical memory errors. To avoid such bugs researchers try to provide both static and dynamic analysis tools and incorporate C/C++ hardening techniques for unsafe Rust. However, such analysis is not always trivial as Rust uses multiple intermediate representations (IR), and when lowering a higher level IR to a lower level IR, all information that marks a specific block of code as unsafe is lost. Since the two worlds of Rust, safe and unsafe, are built on different assumptions many analysis tools and techniques can benefit from knowing where the safe context changes to unsafe and vice-versa.

In this work, we present a modified Rust compiler (*rustc++*) that extends the capabilities of the Mid-level Intermediate Representation (MIR) to enable serialization and metadata enhancements. Our framework introduces MIR++, an enhanced MIR representation that embeds metadata identifying safe and unsafe instructions, enabling fine-grained safety tracking throughout the compilation pipeline. Additionally, *rustc++* enhances the LLVM-IR representation and produced binaries with metadata allowing LLVM passes and post-compilation analysis tools to distinguish between the safe and unsafe context. Lastly, *rustc++* makes preliminary efforts in serializing Rusts MIR to a portable JSON format using the Serde serialization library. Our goal is to allow external tools to work on MIR and its in-memory metadata, without requiring the full compiler pipeline. Modifications of *rustc++* maintain compatibility with the existing Rust ecosystem while offering a powerful foundation for future analysis tools.

## CCS CONCEPTS

• **Security and privacy → Systems security**; **Software security engineering**; • **Software and its engineering → Source code generation**; **Compilers**.

## KEYWORDS

Rust, Rust Compiler Extensions, Borrow Checker, Mixed Binaries, MIR++, Memory Safety


*EuroSec'25, March 30-April 3, 2025, Rotterdam, Netherlands*

ACM ISBN 979-8-4007-1563-1/2025/03…$15.00
https://doi.org/10.1145/3722041.3723102

**ACM Reference Format:**
Antonis Louka, Georgios Portokalidis, and Elias Athanasopoulos. 2025. rustc++: Facilitating Advanced Analysis of Rust Code . In *The 18th European Workshop on Systems Security (EuroSec'25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3722041.3723102

## 1 INTRODUCTION

Compiler-based analysis plays a critical role in identifying performance bottlenecks, security vulnerabilities, and code quality issues. Traditionally, such analysis has focused on C/C++ through various compiler extensions or toolchains. With the introduction of Rust, which implements a unique memory-safety model, new analysis challenges emerged. Rust uses a set of compile-time policies, broadly known as the borrow checker, to achieve memory safety with minimal performance overheads. While the borrow checker provides memory safety, Rust allows unsafe operations to facilitate easier integration with existing C/C++ code. Safe and unsafe code can co-exist in Rust binaries creating a new set of challenges for analysis, and hardening tools on how to handle mixed binaries efficiently. In Listing 1 we provide an example on how the coexistence of the two worlds can affect memory safety.

The pipeline of Rust's compiler (*rustc*) also introduces multiple intermediate representations like HIR (High-level Intermediate Representation) [19] and MIR (Mid-level Intermediate Representation) [21] before lowering to LLVM's IR. *rustc* uses MIR to apply Rust's memory-safety policies and optimizations. While MIR is an ideal representation to take advantage of the Rust specific semantics, it is still missing important information when it gets lowered from HIR. For example, during the HIR to MIR lowering phase, the safety context (instructions residing in unsafe code blocks) and metadata for raw pointers are lost.

Analysts are given two options for analysis when using *rustc*. One approach is to use the *rustc_driver* [18] and *rustc_interface* [20] modules provided from the Rust compiler for accessing the HIR and MIR intermediate representations. These modules provide an API to the Rust compiler allowing analysts to use *rustc* as a library. Tools like Rudra [2], SafeDrop [5], MirChecker [10] utilize this approach to perform static analysis (control-flow analysis) on MIR and discover bugs in Rust code. However, a more traditional approach is to create and add your own passes. *rustc* enables analysts to create and add MIR and LLVM-IR analysis passes for optimizations, hardening and code analysis to the compiler pipeline. This approach is more involved and should be used when there is a need for changing MIR data or for having access to lower IRs than MIR.

Works like ERASan [13] utilize such passes while recreating information that is lost during lowering phases. Specifically, ERASan tries to incorporate a more efficient approach to an address sanitizer (Asan) [32] for Rust binaries by instrumenting only the unsafe

blocks of code. ERASan focuses on the potentially problematic code, and reduces the performance and memory overheads of the original ASan. Works like ERASan, with a goal to meaningfully extend or repurpose traditional analysis frameworks must hijack the Rust compiler pipeline to integrate more metadata information to perform their analysis.

In this work we propose an extended version of the Rust compiler (version 1.83.0), namely *rustc++*, that preserves important metadata for analysis in the *rustc* pipeline. By providing an enhanced MIR version we hope to enable researchers to efficiently adapt existing analysis techniques from the C/C++ domain to Rust and create new ones while incorporating Rust's specific paradigms. This allows analysis to leverage Rust's safety properties, optimize low-level performance and verify security constraints. To foster further research, we will offer *rustc++* as an open-source project, allowing broader adoption and contributions from the community. Currently, the *rustc++* compiler extension provides the following key functionalities:

- We embed extra metadata to MIR; specifically, we introduce MIR++ an enhanced MIR representation that incorporates additional metadata in capturing the origins of each MIR statement (safe/unsafe context). This allows a clear distinction between the instructions originating from safe and unsafe blocks. This metadata is added to the MIR before any other optimization passes take place.
- We carry the MIR++ metadata to the LLVM-IR as instruction metadata. Distinguishing which LLVM-IR instructions are result of unsafe blocks. We then propagate this metadata to the binary by creating a custom section with assembly instructions that are result of unsafe blocks.
- We perform the first steps of MIR serialization. Specifically we use a serialization mechanism to serialize Rust's MIR into a reusable format allowing external tools to perform off-line analysis, and potentially re-loading the MIR metadata back to the compiler at a later stage.

In the remainder of this paper, we detail the design and implementation of *rustc++*, discuss how MIR++ and serialization work, and demonstrate usage of the added metadata.

```
fn get_vec() -> Vec<u8> {
  //create local string variable
  let mut string_v = String::from("deadbeaf");
  //create raw pointer from string
  let mut_raw_ptr = string_v.as_mut_ptr();
  let vector_v;
  unsafe {
    // create a vector variable from string raw pointer
    vector_v=Vec::from_raw_parts(mut_raw_ptr,string_v.len(),
          string_v.len());
  }
  return vector_v; //return vector pointing to freed memory
}
//string_v is dropped, but vector_v is returned
fn main() {
  let vector_var = get_vec();
  println!("{:?}", vector_var);
  //vector_var freed causing a double free
}
```

**Listing 1: Example demonstrating problematic interaction between safe and unsafe context, using raw pointers.**

## 2 BACKGROUND

***Rust language.*** Rust is a modern systems programming language designed to achieve high performance while ensuring memory safety. It enforces memory safety through compile-time checks, eliminating the need for runtime support. To guarantee spatial safety, the Rust compiler (*rustc*) allocates statically sized buffers on the stack. For each access to these buffers, *rustc* generates code to perform bounds checking. Temporal safety is enforced through the "Borrow Checker" [17], a compilation routine that operates on an intermediate representation known as the Mid-level Intermediate Representation (MIR) [21]. MIR serves as the internal layer where Rust applies both optimizations and safety checks before lowering the code to LLVM Intermediate Representation (LLVM-IR). The Borrow Checker enforces Rust's core memory safety guarantees by implementing the language's concepts of ownership, borrowing, and lifetimes. Ownership provides automatic memory allocation and deallocation, borrowing allows the use of references [35], and lifetimes define the scope and duration of variables. These mechanisms ensure memory safety and prevent common errors such as data races and dangling pointers, without runtime overhead.

***Unsafe operations in Rust.*** The *rustc* compiler performs static analysis to verify if code upholds memory safety guarantees. If the analysis is inconclusive, *rustc* prefers to reject valid programs rather than risk accepting unsafe ones. The *unsafe* keyword allows programmers to bypass these restrictions, taking full responsibility for the safety of such operations.

By writing *unsafe* code, the programmer gains access to what Rust refers to as *unsafe superpowers*. These include, dereferencing raw pointers, calling an unsafe function or method, accessing or modifying a mutable static variable, implementing an unsafe trait, and accessing fields of unions [24]. To clarify, it is important to mention that the use of the unsafe keyword does not disable ownership or borrowing concepts enforced by the borrow checker. Instead it permits the programmer to perform these five operations (which are not checked by *rustc*) freely. This places the responsibility for safe memory management on the programmer, where errors can potentially occur.

The use of *unsafe* blocks in Rust is unavoidable, as many operations, such as interacting with hardware APIs or implementing certain data structures like cyclic types (e.g., doubly linked lists) cannot be performed with purely safe Rust. Developers also encounter unsafe code indirectly through both standard (*std, core*) and third-party libraries. According to Rust Foundation as of May 2024 Rust significant crates measure to 123,000 with 24,362 of them using the unsafe keyword (approximately 20% of all crates). Moreover, 35% call other crates that use unsafe keywords, with nearly 20% containing at least one instance of unsafe blocks [1, 34]. The co-existence and interaction of safe and unsafe Rust code, combined with logical programming errors can lead to memory safety vulnerabilities, as seen from works in Rust memory safety (see Sec. 3, and Sec. 6), and entries in Rust Advisory Database [6] that are similar to the example in Listing 1.

***Rust compiler pipeline.*** The Rust compiler pipeline involves multiple lowering stages between Rust's intermediate representations (Figure 1). Notably, the Abstract Syntax Tree (AST) is lowered to High-Level Intermediate Representation (HIR) which closely

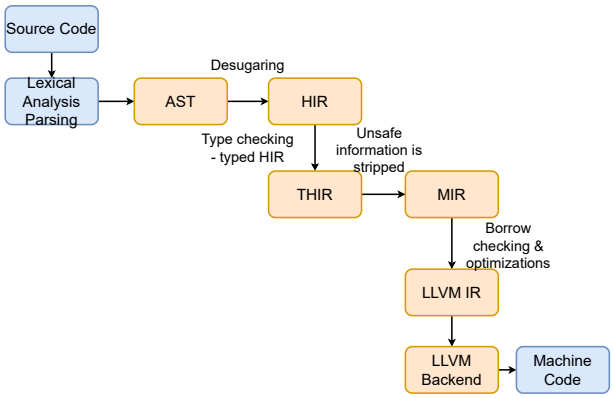


**Figure 1: Rust compiler intermediate representation lowering pipeline**

retains Rust's syntax. After type checking the Rust compiler generates the Typed High-Level Intermediate Representation (THIR) [22], used for MIR construction, exhaustive checks and unsafety checking. THIR is further lowered to MIR, a simpler representation closer to machine-level code. At this stage all information regarding Rust's unsafety context and raw pointers are removed. At the MIR level *rustc* performs control-flow-sensitive analysis (Borrow checking) and enforces the ownership and borrowing rules, while applying optimizations for efficient code generation The MIR is then lowered to LLVM IR, used by the LLVM framework [8] to perform target-specific optimizations. With Rust-specific semantics, like borrow checking completed, LLVM focuses on machine-level optimizations. The LLVM backend converts LLVM IR to machine code, performs final optimizations (e.g., inlining, register allocation), and generates an executable binary.

## 3 CHALLENGES FOR RUST MEMORY SAFETY

Despite Rust's memory safety mechanisms, underlying issues and bugs are still present in Rust binaries. Several studies have demonstrated the existence of different vulnerabilities of Rust, resulting from the way Rust guarantees memory safety, from compiler logic holes (e.g. the CVE-rs GitHub project [33] creates buffer-overflow and use-after-free vulnerabilities by using 100% safe Rust code), and through the Foreign Function Interface (FFI) used for interacting with other programming languages.

Papaevripides et al. [14] demonstrate the existence of attacks that bypass C/C++ hardening techniques such as Control-flow Integrity (CFI) and SafeStack. They emphasize that when dealing with mixed binaries, the interaction of safe and unsafe code weakens the effectiveness of hardening techniques on unsafe code. In similar manner, Mergendahl et al [12] demonstrate how languages with compile-time and runtime safety mechanisms can be affected. The authors explain that safe code (using compile time checks) and unsafe code context (using hardening techniques) operate on different and incompatible assumptions allowing attacks that would be impossible in either context alone to happen when combined.

Currently, *rustc* is not able to pinpoint if any attacks are possible due to the interaction of safe and unsafe code. Official Rust tools like Miri [23] and Rust fuzzing [4] are able to detect some bugs in Rust code using static analysis and fuzzing, but they are not able to cover all cases. Rust's unique memory safety guarantees have inspired several frameworks aimed at enhancing and evaluating its safety. Bang et al. in TRust [3] introduce a framework for in-process isolation of safe Rust code from untrusted components (unsafe blocks), by allocating them to separate memory regions, preventing access to critical data in safe regions.

Other frameworks focus on enhancing Rust's memory safety and debugging capabilities. Min et al. [13] improve the performance of Address Sanitizer (ASan) by identifying raw pointers and selectively instrumenting memory accesses, targeting unsafe regions of Rust programs, addressing both temporal and spatial safety memory bugs. Cui et al. [5] employ a path-sensitive data-flow analysis to detect memory deallocation bugs, such as use-after-free and double free, focusing on unsafe drop calls. Li et al. [10] (MIR Checker) perform static analysis (numerical and symbolic constraint solving) on Rust's MIR to identify runtime crashes and memory-safety bugs. Louka et al. [11] validate Rust's memory safety post-compilation, using static data-flow analysis on safe Rust binaries. They mention that mixed-binary validation is still a challenge due to missing metadata information about the unsafety context.

Some works mentioned so far, modify the Rust compiler to recover information lost during the lowering phases of the intermediate representations used by *rustc*. While other works utilize the rustc_driver and rustc_interface modules. Since many defense and analysis tools rely on this process, we aim to provide a framework that can propagate forward such information for analysis and defense components to use hassle-free.

## 4 IMPLEMENTATION

### 4.1 Metadata propagation

Before discussing the specific changes made to the compiler, we outline the key stages of *rustc*, as shown in Figure 1. The Rust compiler lowers the Abstract Syntax Tree (AST) to its first intermediate representation, HIR. HIR closely resembles the Rust source code, preserving important artifacts and context. At this stage, it is easy to distinguish between instructions belonging to safe and unsafe contexts. However, this information is lost during lowering to MIR. For us this is significant as *rustc* supports optimization passes at the MIR level, many of which could benefit from retaining safe/unsafe metadata. Our objective is to not only maintain this information during the MIR phase but also propagate it to subsequent stages, such as LLVM-IR and the final binary, to facilitate further analysis. To achieve this, we introduce MIR++, an enhanced version of MIR that restores the lost metadata.

***Extending MIR.*** MIR is the main representation in *rustc* where safety-checking and optimizations are performed. To correlate unsafety information from HIR to MIR we locate unsafe HIR statements. HIR is defined as a map module; *rustc* uses *HirIds* which are identifiers that uniquely identify nodes in HIR map, and the visitor trait as the default way of traversing HIR. The visitor pattern is a stateful way of easily traversing heterogeneous data while communicating information. We implement a custom *Visitor* to traverse the HIR map and gather HirIds related to unsafe blocks in a stateful way (storing unsafe HirIds in specific structures for later use). Our

HIR traversal targets unsafe blocks and functions, as ultimately, all unsafe behavior originates from these elements. For instance, implementing an unsafe trait results in either unsafe functions or calls to those functions enclosed within an unsafe block.

We utilize the unsafe HirIds to implement an MIR pass that establishes a relationship between MIR instructions and unsafe HIR instructions. Since there is no existing flag to store this information, we introduce our own. In *rustc*, MIR is organized within a Body object, which includes among other details a list of Basic Blocks (BBLs). Each BBL contains a series of statements and concludes with a terminator. MIR instructions within a BBL are commonly represented by statements [29], while the terminator [31] is a special statement that marks the end of the BBL, defining the next block in the control flow. To retain safety information, we extend Rust's statements and terminators, adding a custom safety flag indicating whether the instruction belongs to a safe or unsafe context.

The exact modifications performed to the MIR components, namely *statements* and *terminators*, to include the safety flag are shown in Listing 2. In practice, we observe that both components share two key fields; *source_info* [27], which records the origin of the MIR entity, and *kind* [30], which defines various kinds of statements and terminators that can appear in MIR. We extend the *SourceInfo* field by adding an *is_safe* flag. During the MIR optimization phase we create a custom pass that runs before other MIR passes, setting the value of this flag for both *statement* and *terminator* fields. The *SourceInfo* struct also includes two additional fields: *span* [28] and *scope*. The *span* field provides information about the instruction's location in the AST and source code, while *scope* holds debugging metadata. To determine the exact location of MIR statements and terminators, we leverage the *span* field in the *SourceInfo* struct. Specifically, we use the unsafe *HirIds* to retrieve the corresponding unsafe *HIR spans*. We then compare the MIR span against these HIR spans and mark the MIR instruction as unsafe if the MIR span is contained within the acquired HIR spans.

```
pub struct Statement<'tcx> {
    pub source_info: SourceInfo,
    pub kind: StatementKind<'tcx>,
}
pub struct Terminator<'tcx> {
    pub source_info: SourceInfo,
    pub kind: TerminatorKind<'tcx>,
}
pub struct SourceInfo {
    pub span: Span,
    pub scope: SourceScope,
    pub is_safe: bool,
}
```

**Listing 2: Modifications done to MIR components: Visual representation of how the SourceInfo struct is modified to incorporate the safety context of MIR instructions.**

***Modifying LLVM-IR.*** To propagate the unsafety metadata to LLVM-IR, we modify the files responsible for lowering MIR to LLVM-IR. In *rustc*, the code handling back-end generation resides in the *rustc_codegen_ssa* crate [25], which defines an interface for different back-ends. Rust supports various back-ends, including LLVM, Cranelift (for WebAssembly), and GCC. For this work, we focus on propagating unsafety metadata to the LLVM back-end, which remains the state-of-the-art framework in modern compiler design. The implementation of code generation for the LLVM back-end is handled by the *rustc_codegen_llvm* crate [26].

We observe that the *rustc_codegen_ssa* module includes a dedicated submodule for generating back-end IR statements from MIR, handled by the *Builder* trait. During compilation, *rustc* iterates through the MIR BBLs, and for each block's statements, it generates various metadata such as debug and coverage information, along with back-end IR instructions. The *Builder* module responsible for lowering MIR *statements* and *terminators* to LLVM-IR instructions is implemented within the *rustc_codegen_llvm* crate, utilizing the LLVM Foreign Function Interface (FFI) integrated into *rustc*.

To propagate the unsafety metadata, we introduce a new flag in the *Builder* dedicated to LLVM-IR generation. This flag is set to reflect the safety context of the MIR *statement* or *terminator* being lowered. We also implement a function within the *Builder* that evaluates the flag each time a new LLVM-IR instruction is generated. To achieve this, we add invocations of the aforementioned evaluation function to every method used by the *Builder* to create LLVM-IR instructions. Based on the evaluation of the safety flag, we generate metadata and attach it to the corresponding LLVM-IR instruction, marking instructions derived from unsafe MIR *statements* or *terminators* as unsafe. It is important to note that this process occurs before LLVM-IR optimizations. As a result, some LLVM-IR instructions, including those marked with unsafe metadata, may be removed during later stages of optimization by *rustc*.

At this stage, we annotate each unsafe LLVM-IR instruction with *unsafe_instr* metadata and perform two additional core operations. First, we correlate these annotated instructions with their corresponding source code using the in-memory debug information generated by *rustc*. This allows analysts using *rustc++* to export the human-readable LLVM-IR format and view both the unsafe instructions and the source code lines that generated them, as shown in Listing 3. Second, we prepare a log file to facilitate the propagation of metadata to the final binary produced by *rustc++*.

```
bb7:      ; preds = %bb6
    %_13 = load i8, ptr %_10.0, align 1, !unsafe_instr !4
...
    br i1 %_14.1, label %panic, label %bb8, !unsafe_instr !4
bb8:  ; preds = %bb7
    store i8 %_14.0, ptr %_10.0, align 1, !unsafe_instr !5
...
!4 = !{!"line:8, col:15, file:src/main.rs"}
!5 = !{!"line:8, col:8, file:src/main.rs"}
```

**Listing 3: Example demonstrating how metadata are depicted in the LLVM-IR textual readable form.**

***Modifying the binary.*** To complete metadata propagation, we enhance the binary by adding a custom section for analysis, containing addresses or ranges of unsafe assembly instructions. Our goal is to allow tools like disassemblers and validators to identify when execution enters an unsafe block. Ideally, this process would be incorporated into the *rustc* backend, but since *rustc* relies on the LLVM framework for machine code generation, we chose to add the metadata post-compilation to avoid modifying LLVM's source code and needing to recompile the framework, using the LLVM precompiled binaries provided by Rust.

In the previous subsection, we explained how *rustc++* generates a log file (in JSON format) during LLVM-IR compilation. This log maps unsafe source code lines to their corresponding files. We then develop a custom disassembler script to load the binary into memory and generate disassembly, mapping assembly instructions

to source code lines using debug symbols. Correlating assembly with source code is complex since a single source line can produce multiple instructions, some of which may originate from external libraries or have no direct relation to high-level code. To address this issue, we perform a function-wide analysis focusing on developer-written code. We gather assembly instructions per function and use debug symbols to map them to source lines, marking them as unsafe, based on the context of those lines. If no mapping is available, we assign the instruction to the context of the last known instruction, assuming the unsafe or safe context persists until a new one is encountered. In simpler terms, if an instruction belongs to an unsafe source line, all subsequent instructions belong to the same (unsafe) context until we reach an instruction that belongs to the safe context.

We note that even if a whole function is marked as unsafe not all assembly instructions are marked as such, this is because our approach associates assembly instructions to the unsafe source code lines. Some assembly instructions such as function prologues or argument handling, may not directly correspond to any high-level code. Finally, we create a custom section named *.unsafe_info* using objcopy, which stores the addresses of unsafe assembly instructions. An example of the output from our script, which emits these unsafe instructions, is shown in Listing 4.

```
0x20c5f: call 0x20ad0  @ line: 5
0x20c64: jmp 0x20c66  @ line: 5
0x20c66: mov rax, qword [rsp + 0x20]  @ line: 0
UNSAFE: 0x20c6b: mov al, byte [rax] @ line: 8
UNSAFE: 0x20c6d: add al, 0xa @ line: 8
UNSAFE: 0x20c6f: mov byte [rsp + 0x17], al @ line: 8
UNSAFE: 0x20c73: setb al @ line: 8
UNSAFE: 0x20c76: test al, 1 @ line: 8
UNSAFE: 0x20c78: jne 0x20ca0 @ line: 8
0x20c7a: mov rax, qword [rsp + 0x20]  @ line: 0
0x20c7f: mov cl, byte [rsp + 0x17]  @ line: 0
UNSAFE: 0x20c83: mov byte [rax], cl @ line: 8
```

**Listing 4: How assembly instructions can be presented using custom disassembler script that performs metadata propagation to the binary**

## 4.2 MIR serialization

In this section we describe our efforts to serialize the MIR representation of the Rust programming language. By serializing MIR, we aim to make it accessible for various tools and analysis without requiring a full Rust compiler run. Our goal is to allow MIR to be represented in a portable format, enabling offline analysis. We aim to repurpose MIR as a reusable and language-agnostic intermediate representation that retains Rust's safety properties and borrow-checking mechanisms, potentially applying them to other contexts like ensuring safety in C/C++ programs.

We serialize the MIR fields in JSON format, using Serde, the de-facto serialization library in Rust. Integration into *rustc++* is achieved by leveraging Serde's directives, such as *#[derive(Serialize)]*, to automatically derive serialization capabilities for necessary data structures (e.g., structs, enums, and unions). Fields that should not be serialized are excluded using the *#[serde(skip)]* directive.

Since manually adding such directives is not feasible, we streamline this process, by creating a script that walks *rustc's* directory structure, loads Rust source code files to the equivalent AST, and recursively visits the AST to automatically add *#[derive(Serialize)]* and *#[serde(skip)]* attributes to all structs, enums, unions and their fields. This automated approach ensures that all structs are prepared for serialization by default, while giving us control to manually enable specific fields; by removing the *#[serde(skip)]* directives. Note that since *rustc* utilizes complex data structures to store MIR, it is sometimes necessary to manually implement the Serialization function for that structure to guide Serde in performing the serialization correctly, requiring additional manual effort from the *rustc++* developers. In this work we annotate all structs and perform preliminary serialization, storing a number of MIR fields and in-memory metadata in the JSON log.

Despite annotating all detected structs, we focus on serializing only MIR-specific metadata. According to the Rust compiler documentation; the MIR of a function is stored in the *Body* struct. Therefore, we enable serialization for fields within this struct. To enable this functionality during compilation, we implement a custom MIR pass that triggers when a specific compiler flag is set. This pass runs after the full MIR is generated and ready for analysis passes. It utilizes the Body struct, serializes its contents to JSON format using Serde, and writes the output to a file (*rustc++* generates one JSON log per MIR body/function). This approach allows users to enable or disable MIR serialization on demand without disrupting the normal compilation pipeline.

# 5 EVALUATION

***Correctness discussion.*** In the *rustc* pipeline, the compiler decomposes source code into an Abstract Syntax Tree (AST), which is then lowered to the High-level Intermediate Representation (HIR) and subsequently to other representations. During each lowering phase, new instructions are generated while older ones are discarded. To maintain the origin information of instructions across these intermediate representations, *rustc* uses *SourceInfo* struct [27], which holds fields such as *span* and *scope*. These fields provide information about the location of each IR instruction in the source code, as explained in Section 4.

By leveraging the *span* field, we can reliably correlate unsafe source code instructions from the AST with their corresponding MIR instructions. To propagate this information accurately to LLVM-IR, we extend the *SourceInfo* struct to include a safety flag that indicates whether a given MIR instruction originates from unsafe code. This modification allows us to focus changes on the *Builder* component, which is responsible for converting MIR instructions to LLVM-IR. The safety flag ensures that metadata are added only when the original MIR instruction is unsafe. Additionally, the generated log file records file names and source code lines for unsafe instructions, enabling precise identification of these instructions during post-compilation analysis. Throughout the compilation pipeline, utilizing the *span* field maintains the relationship between intermediate representation (IR) instructions and the Rust source code, facilitating easy correctness verification of metadata propagation.

***Sample MIR pass.*** In this subsection, we demonstrate an example MIR pass, in this case, a statistical analysis pass, that leverages unsafe metadata added to MIR. This is feasible because *rustc++* identifies and propagates unsafety metadata before any other MIR optimization passes, ensuring availability of this information for all subsequent passes. We integrate this test pass into the *rustc++*

pipeline, executing it before any compiler optimizations. To illustrate this process, we provide a toy example program compiled with *rustc++*, shown in Listing 5.

```
fn main() {
    let mut str = String::from("deadbeef");
    println!("{}", str);
    let ptr = str.as_mut_ptr();
    let _str1 = String::from("deadbeef");

    unsafe {
        *ptr = *ptr + 10;
    }
    let _str2 = String::from("deadbeef");

    println!("{}", str);
    let _str3 = String::from("deadbeef");

    unsafe {
        foo(&mut str);
    }
    let _str4 = String::from("deadbeef");
}
```

**Listing 5: Testing unsafe metadata: Main function of toy example compiled with rustc++**

We present the results of the MIR pass, identifying unsafe MIR instructions using the *unsafety* flag set by *rustc++*. This pass also correlates unsafe MIR instructions with the corresponding source code lines that generated them. The output of this process is shown in Listing 6.

```
Unsafe: StorageLive(_19) @ line 7
Unsafe: StorageLive(_20) @ line 8
Unsafe: _20 = copy (*_14) @ line 8
Unsafe: _21 = AddWithOverflow(copy _20, const 10_u8) @ line 8
...
Unsafe: _41 = &mut _1 @ line 16
Unsafe: _40 = &mut (*_41) @ line 16
Unsafe: AssignOnReturn{return_:[bb14],cleanup:Some(bb22),place
      :Call(_39)} @ line 16
Unsafe: _38 = &mut (*_39) @ line 16
...
Unsafe: StorageDead(_37) @ line 16
Unsafe: _36 = const () @ line 15
Unsafe: StorageDead(_36) @ line 17
```

**Listing 6: Partial result of simple MIR pass: Locating unsafe MIR and their source code lines.**

***Sample LLVM pass.*** We demonstrate the usability of LLVM-IR metadata by implementing a simple LLVM pass. Using *rustc++*, we compile the toy example from Listing 5 and generate the textual representation of LLVM-IR, as shown in Listing 3. This LLVM pass loads the *.ll* file, iterates through functions, and prints unsafe LLVM-IR instructions located in different basic blocks (BBLs) based on the metadata added by *rustc++*. The pass is executed using the LLVM *optimizer* and can identify LLVM-IR instructions marked with *unsafe_instr* metadata. The results of this custom LLVM pass are presented in Listing 7.

```
Function: _ZN7testing4main17h989262e97a945127E
Unsafe: %_13 = load i8, ptr %_10.0, align 1, !unsafe_instr !4
Unsafe: %_14.0 = extractvalue {i8, i1} %8, 0, !unsafe_instr !4
Unsafe: %_14.1 = extractvalue {i8, i1} %8, 1, !unsafe_instr !4
...
Unsafe: br i1 %_14.1,label %panic,label %bb8, !unsafe_instr !4
Unsafe: invoke void @_ZN7testing3foo17h5809e028abd2d08cE(ptr
      align 1 %_25.0, i64 %_25.1)to label %bb15 unwind label %
      cleanup3, !unsafe_instr !7
...
```

**Listing 7: Partial result of LLVM-IR pass run on textual representation with unsafety metadata emmited by rustc++.**

## 6 RELATED WORK

Xu et al. [36] present RPG, a fuzzing tool that enhances fuzzing for Rust. It generates fuzz targets for Rust libraries using a pool-based search using input sequences that prioritize unsafe code paths to uncover bugs related to unsafe code interactions. Bae et al. [2] introduce Rudra, a static analysis tool that detects memory safety issues by analyzing specific code patterns which involve unsafe code blocks leading to vulnerabilities. Kirth et al. [7] address memory isolation between safe and unsafe code, particularly in mixed-language environments. PKRU-safe provides fine-grained heap isolation by leveraging Intel Memory Protection Keys (MPK) and uses profiling-based data-flow tracking to partition memory. Multi-Level Intermediate Representation (MLIR) [9, 15, 16], is a sub-project of the LLVM project; MLIR is developed by Google and aims to help in software fragmentation, and the creation of domain specific compilers. Even though MLIR is still new it is a powerful concept allowing creation and addition of unique intermediate representations in the LLVM framework pipeline, for optimization and analysis passes. MLIR can potentially provide a way for creating compiler intermediate representations and safety analysis routines for currently unsafe languages like C/C++.

## 7 FUTURE WORK

*rustc++* currently incorporates only unsafety metadata. However, based on bugs found in the Rust Advisory Database and insights from Rust research analysis tools, we argue that these works could benefit from including additional information, such as *raw pointer metadata*. Raw pointers are a feature of Rust that allows the creation of C-like pointers. While *raw pointers* can be declared within safe Rust code, they cannot be dereferenced or used outside of an unsafe block. Recent bugs have highlighted that raw pointer variables are a common source of use-after-free errors in Rust, especially since they can be cast back into safe Rust references (see Listing 1).

*MIR Deserialization.* While serialization provides a valuable way to export MIR for external use, our long-term objective is to enable deserialization of MIR back into the Rust compiler pipeline. This task is inherently more complex, as it requires recreating the full MIR structure from JSON and reintegrating it into the compiler pipeline. This would enable MIR to be a truly language-agnostic IR, allowing it to incorporate Rust's safety mechanisms and passes to inherently unsafe programming languages like C/C++.

## 8 CONCLUSIONS

In this paper, we introduced *rustc++*, a Rust compiler extension curated for analysis purposes. We presented *MIR++* an enhanced version of Rust's most important IR, that maintains and propagates unsafety information. This metadata is passed forward through the compilation pipeline, including both LLVM-IR and the final binary, ensuring information is preserved for analysis. Through *rustc++* we made the first steps in transforming *MIR++* to a language-agnostic IR, with goal to offer a foundation for advanced analysis, optimizations tools, and optimal cross-language hardening techniques integration.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback. This work was supported by the European Union's Digital Europe Programme (CyberSecPro, Grant No. 101083594) and Horizon Europe Programme (SecOPERA, Grant No. 101070599). The views expressed in this paper are solely those of the authors and not of the European Commission or project partners.

## REFERENCES

[1] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 2020. How do programmers use unsafe rust? *Proc. ACM Program. Lang.* 4, OOPSLA, Article 136 (Nov. 2020), 27 pages. https://doi.org/10.1145/3428204

[2] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. 2021. Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 84–99. https://doi.org/10.1145/3477132.3483570

[3] Inyoung Bang, Martin Kayondo, HyunGon Moon, and Yunheung Paek. 2023. TRust: A Compilation Framework for In-process Isolation to Protect Safe Rust against Untrusted Code. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 6947–6964. https://www.usenix.org/conference/usenixsecurity23/presentation/bang

[4] Rust Fuzz Book. 2025. Fuzz Testing of Rust code. https://rust-fuzz.github.io/book/cargo-fuzz.html

[5] Mohan Cui, Chengjun Chen, Hui Xu, and Yangfan Zhou. 2023. SafeDrop: Detecting Memory Deallocation Bugs of Rust Programs via Static Data-flow Analysis. *ACM Trans. Softw. Eng. Methodol.* 32, 4, Article 82 (May 2023), 21 pages. https://doi.org/10.1145/3542948

[6] Rust Secure Code Working Group. 2024. RustSec Advisory Database. https://rustsec.org/

[7] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2022. PKRU-safe: automatically locking down the heap between safe and unsafe languages. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 132–148. https://doi.org/10.1145/3492321.3519582

[8] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75.

[9] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[10] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C.S. Lui. 2021. MirChecker: Detecting Bugs in Rust Programs via Static Analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 2183–2196. https://doi.org/10.1145/3460120.3484541

[11] Antonis Louka, Antreas Dionysiou, and Elias Athanasopoulos. 2024. Validating Memory Safety in Rust Binaries. In *Proceedings of the 17th European Workshop on Systems Security* (Athens, Greece) *(EuroSec '24)*. Association for Computing Machinery, New York, NY, USA, 8–14. https://doi.org/10.1145/3642974.3652281

[12] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. 2022. Cross-language attacks. https://www.ndss-symposium.org/wp-content/uploads/2022-78-paper.pdf

[13] Jiun Min, Dongyeon Yu, Seongyun Jeong, Dokyung Song, and Yuseok Jeon. 2024. ERASan: Efficient Rust Address Sanitizer. In *2024 IEEE Symposium on Security and Privacy (SP)*. 4053–4068. https://doi.org/10.1109/SP54263.2024.00258

[14] Michalis Papaevripides and Elias Athanasopoulos. 2021. Exploiting Mixed Binaries. *ACM Trans. Priv. Secur.* 24, 2, Article 7 (jan 2021), 29 pages. https://doi.org/10.1145/3418898

[15] LLVM project. 2024. MLIR: Multi-Level Intermediate Representation. https://mlir.llvm.org/

[16] LLVM project. 2024. MLIR: Multi-Level Intermediate Representation. https://github.com/llvm/llvm-project/tree/main/mlir

[17] rust-borrow checker. 2021. Rust Compiler Development Guide. https://rustc-dev-guide.rust-lang.org/borrow_check.html.

[18] Rust-Lang. 2021. Rust Driver. https://rustc-dev-guide.rust-lang.org/rustc-driver/intro.html#rustc_driver

[19] Rust-Lang. 2021. Rust HIR. https://rustc-dev-guide.rust-lang.org/hir.html

[20] Rust-Lang. 2021. Rust Interface. https://rustc-dev-guide.rust-lang.org/rustc-driver/intro.html#rustc_interface

[21] Rust-Lang. 2021. Rust MIR. https://rustc-dev-guide.rust-lang.org/mir/index.html

[22] Rust-Lang. 2021. Rust THIR. https://rustc-dev-guide.rust-lang.org/thir.html

[23] Rust-Lang. 2024. Miri: an Undefined Behavior tool for Rust (MIR interpreter). https://github.com/rust-lang/miri

[24] Rust-Lang. 2024. The Rust Programming Language: Unsafe Superpowers. https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html#unsafe-superpowers

[25] Rust-Lang. 2025. Rust Compiler Agnostic Backend. https://rustc-dev-guide.rust-lang.org/backend/backend-agnostic.html

[26] Rust-Lang. 2025. Rust Compiler LLVM Backend. https://rustc-dev-guide.rust-lang.org/backend/backend-agnostic.html#refactoring-of-rustc_codegen_llvm

[27] Rust-Lang. 2025. Rust Compiler SourceInfo struct. https://rustc-dev-guide.rust-lang.org/hir.html

[28] Rust-Lang. 2025. Rust Compiler Span struct. https://doc.rust-lang.org/nightly/nightly-rustc/rustc_span/span_encoding/struct.Span.html

[29] Rust-Lang. 2025. Rust Compiler Statement struct. https://doc.rust-lang.org/nightly/nightly-rustc/rustc_middle/mir/struct.Statement.html

[30] Rust-Lang. 2025. Rust Compiler StatementKind struct. https://doc.rust-lang.org/nightly/nightly-rustc/rustc_middle/mir/enum.StatementKind.html

[31] Rust-Lang. 2025. Rust Compiler Terminator struct. https://doc.rust-lang.org/nightly/nightly-rustc/rustc_middle/mir/terminator/struct.Terminator.html

[32] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[33] Speykious. 2023. Blazingly fast memory vulnerabilities, written in safe rust. https://github.com/Speykious/cve-rs.

[34] Rust Foundation Team. 2024. Unsafe Rust in the Wild: Notes on the Current State of Unsafe Rust. https://rustfoundation.org/media/unsafe-rust-in-the-wild-notes-on-the-current-state-of-unsafe-rust/

[35] The Rust Project Developers. 2022. *The Rust Programming Language: References and Borrowing*. Accessed on November 20, 2023.

[36] Zhiwu Xu, Bohao Wu, Cheng Wen, Bin Zhang, Shengchao Qin, and Mengda He. 2024. RPG: Rust Library Fuzzing with Pool-based Fuzz Target Generation and Generic Support. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 124, 13 pages. https://doi.org/10.1145/3597503.3639102