TYPEPULSE: Detecting Type Confusion Bugs in Rust Programs

Hung-Mao Chen[*], Xu He[*], Shu Wang[*†], Xiaokuan Zhang[*], Kun Sun[*]

[*]*George Mason University*
[†]*Palo Alto Networks*

Abstract

Rust supports type conversions and safe Rust guarantees the security of these conversions through robust static type checking and strict ownership guidelines. However, there are instances where programmers need to use unsafe Rust for certain type conversions, especially those involving pointers. Consequently, these conversions may cause severe memory corruption problems. Despite extensive research on type confusion bugs in C/C++, studies on type confusion bugs in Rust are still lacking. Also, due to Rust's new features in the type system, existing solutions in C/C++ cannot be directly applied to Rust. In this paper, we develop a static analysis tool called TYPEPULSE to detect three main categories of type confusion bugs in Rust including misalignment, inconsistent layout, and mismatched scope. TYPEPULSE first performs a type conversion analysis to collect and determine trait bounds for type pairs. Moreover, it performs a pointer alias analysis to resolve the alias relationship of pointers. Following the integration of information into the property graph, it constructs type patterns and detects each type of bug in various conversion scenarios. We run TYPEPULSE on the top 3,000 Rust packages and uncover 71 new type confusion bugs, exceeding the total number of type confusion bugs reported in RUSTSEC over the past five years. We have received 32 confirmations from developers, along with one CVE ID and six RUSTSEC IDs.

1 Introduction

Rust [42] is an emerging programming language known by its strict enforcement of type safety and memory safety through compile-time checking, without sacrificing runtime performance. Since memory safety issues in unsafe languages such as C and C++ have been known to lead to catastrophic consequences, Rust has become an appealing solution to replace C and C++, and it has been adopted in major open-sourced projects such as the Linux kernel [65] and the Firefox browser [35]. Recently, the White House also calls for adoption of memory-safe programming languages such as Rust to secure the cyberspace [23].

Rust is fundamentally divided into two separate sub-languages: safe Rust and unsafe Rust [62]. Safe Rust enforces strict compile-time checks to maintain memory safety and type safety. However, these checks can be excessively restrictive, blocking some essential but risky operations like accessing raw pointers. To address this, Rust introduces the `unsafe` keyword for such situations. When employing unsafe Rust, it falls upon the programmer to uphold memory safety since the typical compile-time checks are circumvented.

Similar to traditional programming languages such as C/C++, Rust also supports type conversions [43], where a variable is initially converted from type *A* to type *B* and subsequently accessed as type *B*. Safe Rust guarantees the security of these conversions through robust static type checking and strict ownership guidelines. Rust automatically infers the types of variables and expressions from their context and use. Also, the ownership feature helps ensure memory and concurrency safety by tracking the lifetime and borrowing at compile time [28]. Nevertheless, there are instances where programmers need to use unsafe Rust for certain type conversions, especially those that involve pointers. For instance, since direct reference conversions are not allowed, conversions must be first made at the raw pointer level and then converted back to references using unsafe Rust. Consequently, these conversions may cause severe memory corruption problems similar to those found in C/C++ [36–38]. In the last five years, RUST-SEC [10] has reported 32 type confusion bugs that can lead to various memory-safety issues such as data leaks, uninitialized memory, and Out-Of-Bounds (OOB) memory access.

Despite extensive research on type confusion bugs in C and C++ [18, 21, 26, 29], studies on type confusion bugs in Rust are still lacking due to three significant challenges. First, the conversion of data types between functions complicates type analysis, and this complexity cannot be addressed using traditional interprocedural analysis. For instance, when a type constructor function creates an instance of a type that is then handed off to another function for an unsafe type conversion, traditional interprocedural analysis fails to track the constructor function as it is absent from the conventional call

graph [31]. Therefore, a new call graph is needed to identify this type of interprocedural type conversion.

Second, predicting all possible concrete types that can replace a generic type in Rust is inherently challenging. Unlike C and C++, Rust uses trait bounds [12] to constrain generic types, ensuring they conform to specific behaviors and capabilities. It adds complexity since trait bounds can have implicit and recursive dependencies on other traits. Moreover, concrete types may encompass composite types like struct. As a result, a method of generic type resolution that adheres to the trait bounds is needed.

Third, identifying type confusion bugs requires establishing whether the pointer alias remains valid after type conversion, which is essential for our bug verification process. However, the ownership and lifetime features in Rust increases the complexity when undertaking pointer analysis. In C++, existing techniques primarily address the issue of pointer access via alias analysis [20, 30, 32]. Nevertheless, traditional alias analysis needs modification to accommodate the pointer variable ownership and lifetime in Rust. For example, when ownership is transferred to another pointer or the pointer is automatically deallocated, the original pointer variable becomes invalid, which cannot be detected by existing techniques.

To tackle these challenges, we develop TYPEPULSE, a static analysis tool to detect type confusion bugs in Rust applications. It consists of two main components, namely, Property Graph Constructor and Bug Detector. By examining each function within the crate, property graph constructor creates a new call graph that helps identify the type constructor functions, addressing the first challenge. Property graph constructor performs type conversion analysis and pointer alias analysis. Type conversion analysis is conducted to collect type pairs (i.e., <source type, destination type>) and trait bounds for generic types through dependency resolution. It is employed to address the second challenge. Pointer alias analysis will construct a new alias graph, representing the alias relationship of the pointers. When ownership is transferred or dropped, property graph constructor updates the alias graph to reflect the node connections, which can solve the third challenge. As the final output, each function in the property graph is associated with type pairs, trait bounds, and the pointer alias graph. The property graph will be utilized by bug detector to analyze and verify the type confusion bugs.

Next, bug detector utilizes the property graph to perform type conversion checks and access checks. First, type conversion checks are used to identify if the type conversion creates an invalid type pointer. Second, access checks will be performed to analyze if the invalid type pointer can be accessed. Via analyzing the type pairs and trait bounds, we summarize three patterns of type confusion bugs, namely, misalignment, inconsistent layout, and mismatched scope, to help locate invalid type pointers. When performing access checks, bug detector will first traverse the alias graph to determine if the invalid type pointer is accessible, then verify the absence

of developer-enforced check, which is manually implemented by developers to prevent the type confusion bugs. After verification, the bug report will be delivered with the unsafe type conversion and type access highlighted.

We implement a prototype of TYPEPULSE with 5249 lines of Rust. To assess TYPEPULSE's effectiveness in bug identification, we first execute it on the type confusion bugs reported to RUSTSEC from 2019 to 2024. The findings show that TYPEPULSE can successfully identify all reported type confusion bugs. Next, we perform a large-scale study by running TYPEPULSE on the top 3,000 popular packages ranked on crates.io and GitHub. We detect 71 new type confusion bugs and report all of them to their package developers. We receive 32 confirmations on the reported bugs at the time of writing. The new bugs occur in many high-profile repositories. For example, we demonstrate that the type confusion bug within the pprof package can cause the downstream applications to crash (e.g., GreptimeDB with 4.2k stars on GitHub).

Contributions. This paper makes the following contributions:

- We analyze all type confusion bugs in RUSTSEC in the last five years and identify the three most prevalent categories of type confusion bugs, namely, misalignment, inconsistent layout, and mismatched scope.

- We design and implement the first static analysis tool (TYPEPULSE) to detect type confusion bugs in Rust, addressing the challenges of interprocedural type conversion, generic type resolution, and alias analysis due to the unique features of Rust.

- We evaluate TYPEPULSE on the top 3,000 Rust packages and identify 71 new type confusion bugs which we have manually confirmed, surpassing the total number of bugs reported in RUSTSEC in the last five years. We also run TYPEPULSE on existing type confusion bugs and it achieves 100% accuracy, demonstrating the robustness of TYPEPULSE.

2 Background

2.1 Rust Basics

Generic Types and Traits. Rust provides the flexibility of code reuse and type safety with generic types and traits [4, 50]. By writing code in a type-agnostic manner, generic types allow functions, methods, or data structures (e.g., struct) to operate on multiple types, which are represented by the placeholder (e.g., T, U). Since the generic types will be replaced by the actual types at the compile-time (i.e., monomorphization), it can prevent the type confusion bugs at run-time. In addition, traits can be used to specify the constraints on generic types. By applying a trait to a generic type, Rust ensures that the type used to replace the generic type should also implement the required methods or characteristics defined by the trait.

For example, the function `fn display<T: Copy>(input: T)` ensures that only the type implementing the `Copy` trait can be initialized as the argument `input`.

Safe vs. Unsafe Rust. The central concept of safe Rust is to confirm memory *ownership* during compilation, where the compiler checks both the access and the *lifetime* of memory-allocated objects (or values). Moreover, safe Rust permits the *borrowing* of a value (i.e., making a reference to it) throughout the lifetime of the owner variable. In contrast, `unsafe` is used to highlight code segments that perform tasks that are not ensured by the compiler, placing the responsibility on developers to prevent memory safety issues. In Rust, there are five specific actions necessitating the `unsafe` keyword [63]: dereferencing raw pointers, invoking unsafe functions, altering or accessing mutable static variables, defining unsafe traits, and executing inline assembly. Each of these actions could breach Rust's safety assurances. Unsafe Rust is crucial because it allows developers to interface with low-level system APIs, libraries written in other languages, or hardware directly.

Undefined Behaviors. Undefined behavior (UB) refers to the program whose outcome is not prescribed by the language's specification, which means that the language standard does not define what should happen if the UB occurs. In most cases, the result can only be decided by hardware and architectures, leading to inconsistent consequences in different environments. The outcomes of UB are unpredictable, ranging from security vulnerabilities to incorrect compiler optimization and code generation. The backend of the compiler might perform the optimization based on the assumption that the UB will not occur. Therefore, we should prevent UB from happening. Rust clearly defines scenarios that might trigger undefined behaviors [64], such as dereferencing null pointers, accessing out-of-bounds array elements, and data races when mutating shared data without synchronization. The design of `unsafe` help us narrow down the culprit of UB to the code related to unsafe code.

2.2 Type Conversion in Rust

Type conversion from the source type (`src_ty`) to the destination type (`dst_ty`) consists of two steps: ❶ *Conversion*, which involves altering or reinterpreting the bit pattern of a variable from one type (`src_ty`) to a new type (`dst_ty`), and ❷ *Access*, which involves accessing the variable as the new type (`dst_ty`). `rustc` limits developers to performing only explicit type conversions to maintain safety with compile-time verifications. These conversions can be implemented through type casting, transmute operations, and traits [50]. Given that traits are typically handled by casting and transmute methods, our discussion will focus solely on casting and transmute operations.

Casting Operation. The type-casting operation depends on the keyword `as`, which is mainly used for secure and direct

```rust
1  fn main() {
2      let source_ty: u8 = 1;
3      // compile error: non-primitive cast
4      let dest_ty = &source_ty as &u32;
5      // Alternative 1: as
6      let tmp_ty = &source_ty as *const u8 as *const u32;
7      let dest_ty = unsafe {&*tmp_ty};
8      // Alternative 2: transmute
9      let dest_ty = unsafe {
10         transmute::<&u8, &u32>(&source_ty)
11     };
12 }
```

Listing 1: Type conversion between pointers in `unsafe` code.

type conversions, including conversions between basic data types and raw pointers. The use of `as` generally involves straightforward bit manipulations or adjustments in values. For example, when an `f32` is converted to an `i32`, the fractional component of the floating point number is removed. Consequently, using `as` can result in data truncation or loss. It's important to note that under the stringent regulations established by `rustc`, `as` can be employed in the `safe` code.

Transmute Operation. Compared to `as`, the `transmute` function facilitates more intricate and dangerous transformations. Essentially, `transmute` performs a bitwise copy from one type to another without altering the bit pattern of the value. However, it modifies the interpretation of these bits by `rustc`. For instance, it allows for the direct conversion of an `i32`'s bit pattern to an `f32`, despite their fundamentally different representations. The validity of the original bit pattern in the new type is not assured, making `transmute` extremely risky and prone to causing undefined behaviors. Consequently, `transmute` necessitates the use of `unsafe` code.

Type Conversion between Pointers. In Rust, pointer-type conversions can be achieved through casting and the use of transmute operations. However, Rust imposes various restrictions depending on the pointer types involved, meaning that some conversions are safely handled by Safe Rust, while others require the use of the `unsafe` keyword. For example, in Listing 1, Safe Rust prohibits direct conversion between reference types (line 3), forcing developers to resort to two methods within `unsafe` code. The first method involves converting the reference of the original type to a raw pointer, followed by its conversion to another raw pointer (line 5). To acquire a reference of the new type, developers must first dereference the raw pointer and then form a new reference (line 6). As dereferencing a raw pointer is not allowed in Safe Rust, the use of `unsafe` code becomes unavoidable. Alternatively, the `transmute` function can be used directly to convert references (line 9), which must be used with the `unsafe` keyword due to its inherent risks. Such conversions in Unsafe Rust are prone to type conversion errors because the memory

Table 1: The details of 32 reported Type Confusion Bugs on RustSec advisories (2019-2024).

Year	Type I	Type II	Type III	Others
2019	2019-0035	-	2019-0028	-
2020	2020-0035 2020-0050	2020-0029 2020-0078 2020-0079 2020-0080 2020-0081	2020-0029 2020-0165	2020-36317 2020-0073 2020-0164
2021	2021-0120 2021-0121 2021-0145	2021-0021 2021-0035	2021-0019 2021-0089	2021-0044
2022	2022-0041	2022-0052 2022-0074	2022-0092	2022-0034 2022-0078
2023	-	-	2023-0015 2023-0055	-
2024	-	2024-0347	2024-0001	-

address remains the same for both pointers (`&src_ty` and `&dst_ty`), while only the interpretation of the type changes.

3 Overview

3.1 Motivating Examples

We investigate all bug reports related to type confusion bugs in the RUSTSEC advisories in the last five years [10]. There are 32 reports but only 26 type confusion bugs; The remaining six bugs are out of consideration because they are related to other memory safety problems, such as Use-After-Free, which can be handled by existing tools [15, 17, 33, 44]. The Bug IDs are listed in Table 1. We categorize the 26 bugs into three types based on their root causes, which are misalignment, inconsistent layout, and mismatched scope. All 26 bugs are related to pointer type conversion (i.e., references and raw pointers) in Unsafe blocks. In this section, we introduce the three bug types and their security impacts.

Type I: Misalignment Bug. The first type of bug occurs when the type is converted to another type leading to alignment violation. The alignment of a type specifies the valid memory address at which the type should be stored. Given the alignment value as `n`, the type must be stored only at the address of a multiple of `n`. Some types have a fixed alignment regardless of the target architectures, while others could be platform-specific. For example, the type of `i32` has both 4-byte alignments on the 32-bit or 64-bit target. In contrast, the types of `usize` and `isize` are aligned to 4 bytes on the 32-bit target and 8 bytes on the 64-bit target. The alignment requirement can be easily violated with pointer-type conversion since two pointers remain in the same memory address. Although the memory address is aligned for `src_ty`, it might not be aligned for `dst_ty` if not handled carefully. In the Listing 2, the method `fill_bytes` allows the slice of `u8` to be cast onto the slice of `u32` (line 9). Since `u8` is aligned to 1 byte, the slice of `dest` can be stored at the arbitrary memory address. When

```rust
1  #[cfg(any(target_arch = "x86",
2      target_arch = "x86_64"))]
3  fn fill_bytes(&mut self, dest: &mut [u8]) {
4      // ...
5
6      while filled < end_direct {
7          let dest_u32: &mut R::Results = unsafe {
8              &mut *(dest[filled..].as_mut_ptr() as
9              *mut <R as BlockRngCore>::Results) };
10         self.core.generate(dest_u32);
11         filled += self.results.as_ref().len() * 4;
12         self.index = self.results.as_ref().len();
13     }
14     // ...
15 }
```

Listing 2: A misalignment bug in `rand_core` that casts bytes slices to integer slices (RUSTSEC-2019-0035 [54]).

it turns out to be accessed as `u32`, it is not guaranteed that the memory address can be multiple of 4 since `u32` is aligned to 4 bytes, leading to the misaligned pointer dereference. Note that developers of `rand_core` consider that the issue could be avoided by limiting the target architectures to `x86` or `x86_64` only since these architectures are designed to tolerate misaligned memory access. However, the alignment requirement is enforced by the compiler instead of these target architectures. Once the Rust compiler verifies that safe code adheres to alignment rules, it generates optimized machine code based on this assurance. However, if unsafe code violates these rules, it can cause undefined behavior or crash the program.

Type II: Inconsistent Layout Bug. The second type of bug occurs when `src_ty` and `dst_ty` have different memory layouts. In Listing 3, the method `as_ref` allows casting between `Table` and `TableSlice` and returns the reference to the new type (line 20). However, when the `struct` type in Rust inherits the default representation (e.g. `repr(Rust)`), the compiler may reorder the memory layout, such as the fields of `struct`. The results of GDB show that after the raw pointer to `Table` is converted to the `TableSlice`, the fields `rows` of `Table` and one of `TableSlice` point to different memory addresses (line 26 and line 28), leading to inconsistent lengths of `rows` (line 27 and line 29). It could impact applications that rely on the value (e.g., `rows.length`). For example, when applications plan to print all the data stored in table format to the terminal, the API (`TableSlice::print_tty`) converts `Table` to `TableSlice` first and iterates the data stored in `rows`. Since iterating slice relies on the length (line 29) while the number of elements is actually only one (line 27), printing table leads to invalid memory access and segmentation fault, which has been reported in RUSTSEC-2022-0074.

Type III: Mismatched Scope Bug. The third type of bug occurs when we break the invariant by creating an invalid bit pattern or modifying the mutability of types. In Listing 4, the trait `ComponentBytes` is designed to provide a

```
1  pub struct Table {
2      format: Box<TableFormat>,
3      titles: Box<Option<Row>>,
4      rows: Vec<Row>,
5  }
6
7  pub struct TableSlice<'a> {
8      format: &'a TableFormat,
9      titles: &'a Option<Row>,
10     rows: &'a [Row],
11 }
12
13 impl<'a> AsRef<TableSlice<'a>> for Table
14 fn as_ref(&self) -> &TableSlice<'a> {
15     unsafe {
16         let s = &mut *(
17             (self as *const Table) as *mut Table
18         );
19         s.rows.shrink_to_fit();
20         &*(self as *const Table as *const TableSlice<'a>)
21     }
22 }
23
24 // from gdb results
25 // $8 as &Table, $7 as &TableSlice
26 p &$8.rows  // 0x7ff..82f0
27 p &$8.rows.len  // 1
28 p &$7.rows  // 0x7ff..82e0
29 p $7.rows.length  // 93825009397280!
```

Listing 3: An inconsistent layout bug in `prettytable-rs` that casts a `&Vec` to `&[T]` (RUSTSEC-2022-0074 [56]).

```
1  pub trait ComponentBytes<T: Copy + Send + Sync + 'static>
2      where Self: ComponentSlice<T> {
3      fn as_bytes_mut(&mut self) -> &mut [u8] {
4          let slice = self.as_mut_slice();
5          unsafe {
6              slice::from_raw_parts_mut(
7                  slice.as_mut_ptr() as *mut _, ..)
8          }
9      }
10 }
11
12 impl<T> ComponentSlice<T> for [RGB<T>] { .. }
13
14 impl<T> RGB<T> {
15     pub const fn new(r: T, g: T, b: T) -> Self {
16         Self {b, g, r}
17     }
18 }
19
20 // exploit for type III bug
21 let component: &'static str = "Hello, World!";
22 let new_rgb = RGB::new(component, .., ..);
23 let mut rgb_arr = [new_rgb; 3];
24 let bytes = rgb_arr.as_bytes_mut();
25 bytes[0] += component.len() as u8;
26 // now, we can modify static memory
27 println!("{}", rgb_arr[0]);
```

Listing 4: A mismatched scope bug in `rgb` that allows viewing and modifying data of any type wrapped in `ComponentSlice<T>` as bytes (RUSTSEC-2020-0029 [55]).

method `as_bytes_mut` to modify the type `T` as byte slice. The type `T` could be any type that implements the traits `Copy`, `Send`, `Sync`, and lifetime bound `static`. However, the trait and lifetime bounds here cannot prevent the issues caused by the problematic type conversion implemented in `as_bytes_mut`. It allows casting between mutable raw pointers (`slice.as_mut_ptr() as *mut u8`) to create an invalid state for types since two pointers are pointing to overlapping memory. Safe Rust enforces aliasing rules, where mutable and immutable references can not exist simultaneously, while unsafe Rust allows the rule to be bypassed, as shown in the exploit (line 21 - 27). The attacker creates an immutable reference pointing to the static string as the type `T`. With `as_bytes_mut`, the mutable raw pointer to the slice of string casts to the mutable raw pointer of `u8` type. Since the function returns a mutable reference to a slice of `u8`, the attacker is allowed to modify any values in the slice of `u8`. However, the mutable reference `bytes` and immutable reference `component` point to the same data, breaking the aliasing rules of safe Rust. While the attacker modifies the value in `bytes`, he also changes the value of `component`, which should not be mutated originally. One security consequence of modifying immutable data is data races. In a multi-threaded environment, if the immutable object can be modified through a mutable reference while other threads are reading it, the outcome could be unpredicted or even lead to a program crash. In addition, applications usually rely on the static variable for security checks or maintaining a global state, which means that mutating the immutable data can also help attackers bypass security checks.

3.2 Challenges and Insights

The Rust type system's features, such as ownership, trait bounds, and generic types, present challenges that cannot be directly addressed using existing methods in C and C++. We use the example of the mismatched scope bug to illustrate the three challenges.

Interprocedural Type Conversion. In Listing 4, TYPE-PULSE identifies that the `as_bytes_mut` function performs a risky type conversion from a generic type to `u8` on line 7, potentially leading to a type confusion bug. To confirm the presence of this bug, TYPEPULSE must determine if the type is converted between functions. Line 3 indicates that type `self` is initialized by a constructor function then passed to the current function. However, finding the constructor is a challenging problem that traditional call graphs cannot address. For instance, the type of `[RGB<T>]` implements `ComponentSlice` (refer to line 12) but must be initialized via the `new` function shown on line 15. Traditional call graphs can locate callers of `as_bytes_mut`, but the type constructor is not typically a direct caller (see lines 22 and 24). To address this issue, we

identify the constructor by matching converted types to the return types of external functions via a new data structure called Property Graph.

Generic Type Resolution. To predict the concrete types that can initialize generic type `T`, we first analyze trait bounds. The generic type `T` is constrained by four trait bounds: `Copy`, `Send`, `Sync`, and `'static`. Suppose we only enumerate the types directly involved in these trait bounds; we only get some internal types, such as `RGB<ComponentType>` and `BGR<ComponentType>`. Since `ComponentBytes` is a public trait, the user can initialize it with external user-defined types, such as `str` for the generic type (line 21). To address such an issue, we extend *type conversion analysis* to resolve the trait bounds. First, if these traits are also bounded by other traits, we need to parse the dependencies recursively. Second, if the function is public, we must consider all the primitive types and composite types that external users could initialize. Therefore, we collect traits and primitive types defined in standard libraries. For the composite types, we build the `struct` type from primitive fields. Finally, bug detector can leverage the trait bounds to generate a type set. The type set includes all possible types that may be implicitly implemented and satisfy the trait bounds.

Alias Analysis. To verify the existence of the bug, it is crucial for us to determine whether the pointer alias remains valid after the type conversion (whether we can access `rgb_arr` after line 24). The pointer type conversion on line 7 is translated to `_8 = move _9 as *mut u8 (PtrToPtr)` in the Mid-level Intermediate Representation (MIR [6]) of the Rust compiler. `move` represents the transfer of ownership of a value to another, which means `_9` is not accessible anymore. However, the previous alias of `_9` still point to the same memory address. Therefore, `slice` on line 4 is still accessible after the ownership of its mutable pointer (`_9`) is transferred, leading to mismatched scope bug when accessing `rgb_arr[0]` (line 27). To precisely identify the alias relationship between the pointers, we analyze whether the pointer's ownership is transferred based on different forms of instruction in the Rust program. In the Listing 4, pointer alias analysis helps us verify that the parameter (`&mut self`) points to the same memory location as the `u8` pointer (`&mut [u8]`), and whether the parameter remains accessible after returning.

3.3 Detection Scope

We target the type confusion bug arising from the pointer type conversions and specify the type conversion behavior to be implemented with `as` and `transmute` operations, representing the most fundamental ways to conduct type conversion in Rust. In particular, TYPEPULSE focuses on the three most prevalent bug types mentioned in §3.1. We do not consider the type conversion performed on non-pointer types, so the bugs such as integer overflow [57, 58] arising from downcast are excluded. Errors arising from foreign function interfaces [3]
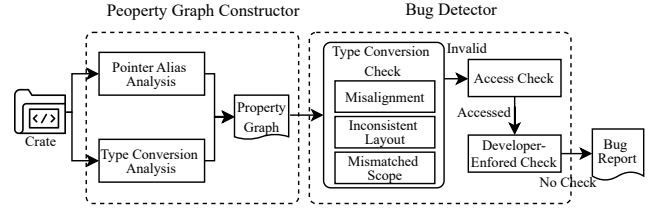

Fig. 1: An overview of TYPEPULSE.

are also out of scope, as addressing them would require developing a system that is compatible with other programming languages' compilers.

4 TYPEPULSE

To facilitate the detection of type confusion bugs, we design and implement a static analysis tool called TYPEPULSE. This tool consists of two main components: Property Graph Constructor and Bug Detector (see Fig. 1). Given the Rust code, property graph constructor first utilizes the compiler to translate the source code into the MIR. It then performs type conversion analysis and pointer alias analysis to construct property graph. The property graph includes the type conversion pairs and the pointer alias graph. Besides, trait bounds are provided to bug detector for generic type resolution. Given the type conversion pairs, bug detector first performs the type conversion check with three different patterns to capture the invalid type pointer, Then, the access check is run to determine whether the invalid type pointer can be accessed through the alias graph. Finally, bug detector verifies if there are any developer-enforced checks implemented to handle the invalid type pointer. If no, TYPEPULSE produces a bug report with the problematic type conversion highlighted.

4.1 Property Graph Constructor

The goal of property graph constructor is to collect the required information and then integrate them into the property graph, which can accelerate the interprocedural analysis of bug detector. In addition to the results of type conversion analysis (§4.1.1) and pointer alias analysis (§4.1.2), each function is associated with its return types, assisting in finding the type constructor functions. For example, given a function `new_as_slice`, it calls the type constructor function `new` and passes the constructed type to the function `as_slice` as `src_ty`. To obtain `new` function, we match `src_ty` with the return types of other functions in our property graph, which could find all potential type constructors. After that, we can further analyze the type conversion across the functions. In addition to return types, the function including `unsafe` code will be marked for analysis in bug detector.

Algorithm 1 `get_trait_bounds()`
Input : *fn, trait_map, visible*
Output : *trait_bounds*
trait_bounds ← HashSet::new();
foreach *trait_bnd* ∈ *fn*.get_bnd_by_sig() **do**
    if *trait_bnd* ∈ *trait_map* **then**
        | *trait_bounds*.insert(*trait_bnd*);
    end
    else
        if *trait_bnd*.has_supertraits()&&*visible* **then**
            | // call get_trait_bounds() on supertraits again
        end
        else
            | *trait_bounds*.insert(*trait_bnd*);
        end
    end
end
return *trait_bounds*;

4.1.1 Type Conversion Analysis

The type conversion analysis has three steps, namely, analyzing if type conversion includes any generic type, determining if generic type is converted across functions, and resolving the dependencies to collect the trait bounds on generic types. In the first step, property graph constructor directly keeps the concrete type pair if no generic type is included. It starts with visiting the MIR's statements [52] and finding the ones of type conversion. Rust's MIR is a simplified version of the Abstract Syntax Tree (AST) used for optimization. It consists of statements and terminators [53]. Statements represent intermediate operations such as assignments and variable initialization, and terminators define control flow decisions such as conditions or function calls. In the statement of type conversion (`src_ty`, `dst_ty`), if both `src_ty` and `dst_ty` are the concrete types, property graph constructor will keep the type pair directly. If one of the (`src_ty`, `dst_ty`) is a generic type, property graph constructor will move on to the second step, which is visibility analysis.

Visibility Analysis. The visibility of a function decides how external users can call the function. In Rust, functions and methods are both blocks of reusable code. The difference between a function and a method is that the method is associated with a particular type or defined within a trait. It is typically called using the "`.`" operator on the type instance. If the generic type conversion occurs in the method, we should determine whether the associated types can only be initialized by type constructor functions. In such cases, we analyze the visibility of all associated types and recursively traverse the types fields if it is a `struct` type. The visibility result represents if the type can be initialized by external users or limited by constructor functions. If visible, trait bound analysis will be conducted to collect the type constraints for the generic type.

Trait Bound Analysis. We collect a set of traits from standard libraries, which are implemented by all primitive types in Rust, indicating the potential concrete types to replace generic types (see algorithm 1). We also extract certain traits from external libraries used to prevent type confusion bugs, helping to reduce false alarms in bug detection. For example, the trait `plain` [40] is always used to ensure that the memory layout is stable and initialized. As we have confirmed the specific types implementing these traits (`trait_map`), we utilize them as the endpoint of the traversal, effectively tackling the issue of implicit dependencies. For each trait bound, property graph constructor first checks whether the trait is defined in the trait set. If not defined, property graph constructor then checks whether the trait has dependencies (`has_supertraits()`). The output of this step also generates the type conversion pairs including generic type and associated with the trait bounds.

4.1.2 Pointer Alias Analysis

Pointer alias analysis is used to construct an alias graph, which helps us determine the relation between pointers and how the pointer can be accessed (see algorithm 2). The analysis is performed in MIR for semantic information [6], e.g., whether a value is moved or borrowed and if the value is dead. The nodes in the alias graph are collected from the `Local` in the MIR, which refers to the "variables and temporary values in the scope of function" [51]. The edges between the nodes are updated when the MIR statement is in the forms of `StorageDead` and `Assign` form, where `Rvalue` is assigned to `Lvalue`. Based on the kinds of `Rvalue` appearing in the statement of `Assign`, pointers have different alias relationships.

$$
\begin{aligned}
a &= \text{Ref(b)} \\
&= \text{RawPtr(b)} \\
&= \text{Cast::(PtrToPtr, Operand(b))} \\
&= \text{Cast::(Transmute, Operand(b))}
\end{aligned} \tag{1}
$$

In Equation 1, when the kind of `Rvalue` is `Ref` or `RawPtr`, which means a new reference or raw pointer a is created and points to the same memory location as b. If the kind of `Rvalue` is `Cast`, especially on the pointers, a also points to the same location as b. In our alias graph, we will create the edge from a to b to represent the alias relationship, where they are both `local`. However, we disconnect the edge from a to b when the kind of `Operand` in `Cast` is `Move`. The operand of `Move` means that the ownership of b is transferred to a, and b will no longer be accessible, so we disconnect the edge. In addition to the statement in the `Assign` form, we also disconnect the edge in the form of `StorageDead`. Given `StorageDead(a)`, it is used to mark that the ownership of a is transferred and all pointers of a become invalid. Therefore, we delete all edges created from a in our alias graph.

$$
a = \text{Call(Fn, args<Operand(b)>, ..)} \tag{2}
$$

Equation 2 presents a function call in MIR, where `args` works as a list of arguments that are passed to the function and a holds the return value. For each argument in `args`, we create the edge from a to b, but disconnect the edge if the operand on

Algorithm 2 `get_alias_graph()`

Input : fn
Output : *alias_graph*
foreach $st \in fn.statements$ **do**
    if $st \in Assign$ **then**
        $(lval, rval) \leftarrow (st.lvalue(), st.rvalue())$;
        $op \leftarrow rval.$get_operand$()$;
        $kind \leftarrow st.rvalue().kind()$;
        if $kind \in Ref|RawPtr|Cast :: PtrToPtr|Transmute$ **then**
            // insert $rval.$id$()$ to alias_graph[$lval.$id$()$]
            if $op == Move$ **then**
                // delete $rval.$id$()$ from alias_graph[$lval.$id$()$]
            end
        end
    end
    else if $st \in StorageDead$ **then**
        $rval \leftarrow st.rvalue()$;
        // delete all elements from alias_graph[$rval.$id$()$]
    end
end
foreach $tm \in fn.terminators$ **do**
    $kind \leftarrow tm.kind()$;
    if $kind ==$ `Call`$(func, args, dest)$ **then**
        foreach $arg \in args$ **do**
            // insert arg.id() to alias_graph[$dest.$id$()$];
            $op \leftarrow arg.$get_operand$()$;
            if $op == Move$ **then**
                // delete $arg.$id$()$ from alias_graph[$dest.$id$()$]
            end
        end
    end
end
return *alias_graph*

---

the argument is `Move`. In bug detector, the connection of `a` and `b` in the alias graph is leveraged to perform interprocedural alias analysis. , the *alias_graph* is constructed as a directed graph where the edge always starts from the `local` in `lvalue` to the one in `rvalue`. When identifying pointer aliasing, we will check whether two nodes have common descendents in *alias_graph*. Finding the common descendent represents that one alias of the `src_ty`'s pointer is aliased with the `dst_ty`'s pointer, then bug detector will collect all descendants while traversing the graph with breadth-first search [66] from two nodes, then find whether there is an intersection between two sets of descendants.

4.2 Bug Detector

Bug detector focuses on the marked functions in property graph (with `unsafe`), capturing type confusion bugs in three steps. First, given the pairs of type sets generated by type conversion analysis, type conversion check is performed to find the invalid type pointer following three kinds of bug patterns. Second, access check is used to find the alias of the invalid type pointer based on pointer alias analysis. Based on the alias graph, it checks if the pointer is accessed in the function or accessible to the caller function. Third, verifying the absence of developer-enforced check helps reduce the false alarms

of bugs. All three steps are combined with interprocedural analysis based on property graph.

4.2.1 Type Conversion Check

We categorize the type conversion (*src_ty*, *dst_ty*)into three possible scenarios: (Con → Con), (Con → Gen), and (Gen → Con). The conversion between (Gen → Gen) is excluded since we observe that such a conversion would be rejected by the `TypeId` check [13]. The check strictly requires two types sharing the same layout. When generic type conversion errors are identified, the trait bounds linked to the generic type are mapped to the *ty_set*, which has been verified to implement these traits in property graph constructor. The detection logic for each bug type and each scenario is shown in Table 2.

Misalignment Detector (Type I). Misalignment detector can easily compute the alignments to identify the bugs; however, it is not possible to predict the alignment of generic types that depend on runtime input. To solve the challenge, we will use *ty_set* to simulate the input to generic types.
Bug Definition. When *src_ty*'s alignment is not a multiple of *dst_ty*'s alignment, it will create a misaligned pointer.
Type Conversion. In the scenario of (Con → Con), misalignment detector directly locates the type conversion by computing the violation of alignment requirements (i.e., `src_ty`.align % `dst_ty`.align != 0), and then we will mark the *dst_ty*'s pointer as an invalid type pointer. In the scenario of (Con → Gen), we need to traverse all candidate types in *ty_set* to ensure each type obeys the alignment requirements. If any candidate type violates the requirement, we mark it as an invalid pointer. When *ty_set* is empty, it means that the generic type can be initialized with arbitrary types since no trait bounds are found. In this case, we will also mark *dst_ty*'s pointer as an invalid pointer. In the scenario of (Gen → Con), the detector follows the same logic as in (Con → Gen) to mark the invalid pointer. The difference is that *dst_ty* can not be aligned to only one byte even when the *ty_set* is empty. The reason is that any memory address can be a multiple of one where the misaligned pointer will not be created. In some cases, misalignment detector may fail since the types are imported from external packages. To solve the challenge, we heuristically extract the information from the symbol names of types based on the cases we have studied (e.g., extract `u8` from `external::u8_bytes`). Since we only run our tool on the machine of 64-bit architecture; however, some type's alignment is platform-specific, where the value changes based on different architectures. Take `usize` and `isize` for example, on a 32-bit target, they are aligned to 4 bytes while on a 64-bit target, they are aligned to 8 bytes. In Misalignment Detector, we will consider different alignment values for these types in the type conversion.

Inconsistent Layout Detector (Type II). To detect the inconsistent layout bug, we define two type sets: *unstable_ty* and *stable_ty*. *unstable_ty* represents the type that can change the

Table 2: Type conversion checks.

Bug Type	Con → Con[‡]	Con → Gen	Gen → Con
Type I	**Input**: *src_ty*, *dst_ty* **If** *src_ty*.align % *dst_ty*.align != 0 mark	**Input**: *src_ty*, *ty_set* **If** *ty_set*.is_empty() mark **Else** replace *dst_ty* with each in *ty_set* run again in (Con → Con)	**Input**: *dst_ty*, *ty_set* **If** *ty_set*.is_empty() & *dst_ty*.align != 1 mark **Else If** ty_set.not_empty() replace *src_ty* with each in ty_set run again in (Con → Con)
Type II	**Input**: *src_ty*, *dst_ty* **If** *src_ty* → *unstable_ty* **If** *dst_ty* → (*stable_ty* | *unstable_ty*') mark	**Input**: *src_ty*, *ty_set* **If** *ty_set*.is_empty() **If** *src_ty* → *unstable_ty* mark **Else** replace *dst_ty* with each in *ty_set* run again in (Con → Con)	**Input**: *dst_ty*, *ty_set* **If** *ty_set*.is_empty() **If** *dst_ty* → (*stable_ty* | *unstable_ty*') mark **Else** replace *src_ty* with each in *ty_set* run again in (Con → Con)
Type III	**Input**: *src_ty*, *dst_ty* **if** *src_ty* → *weak_ty* **If** *dst_ty* → *strict_ty* mark **Else If** *src_ty* → *strict_ty* **If** *dst_ty* → *mut weak_ty* mark	**Input**: *src_ty*, *ty_set* **If** *ty_set*.is_empty() **If** *src_ty* → *weak_ty* mark **Else If** *src_ty* → *strict_ty* && *mut dst_ty* mark **Else** replace *dst_ty* with each in ty_set **If (s,d)**[*] → (*weak_ty*, *strict_ty*) mark **Else If (s,d)** → (*strict_ty*, *mut weak_ty*) mark	**Input**: *dst_ty*, *ty_set* **If** *ty_set*.is_empty() **If** *dst_ty* → *strict_ty* mark **Else If** *dst_ty* → *weak_ty* && *mut dst_ty* mark **Else** replace *src_ty* with each in *ty_set* **If (s,d)** → (*weak_ty*, *strict_ty*) mark **Else If (s,d)** → (*strict_ty*, *mut weak_ty*) mark

[‡] Con: concrete type; Gen: generic type; [*] (s,d): (*src_ty*, *dst_ty*).

memory layout at runtime (e.g., struct, union, trait object), where the compiler preserves the rights to insert padding bytes or reorder the fields. Another type set *stable_ty* consists of scalar types (e.g., bool, char, integers). Inconsistent Layout Detector will perform further analysis on the representation of types (e.g., repr(Rust), repr(transparent), repr(C)). Any type conversion in *unstable_ty* set or across *unstable_ty* and *stable_ty* sets would be recognized as a problematic type conversion and create an invalid type pointer. In addition, we need to combine with *ty_set* to extend the scenarios of generic type conversion.

Bug Definition. When the layout of *src_ty* is not stable and inconsistent to *dst_ty*, it will create an invalid type pointer.

Type Conversion. If the detector finds the conversion happens in (*unstable_ty* → *stable_ty*), it will mark the *dst_ty*'s pointer as an invalid type pointer since the padding bytes can be exposed when we accessed them as a scalar type. The second pattern is (*unstable_ty* → *unstable_ty*), inconsistent layout detector will further check if they follow the same Application Binary Interface (ABI) [2], which determines if *src_ty* and *dst_ty* share same layout based on their symbol name of type. If they have different type symbol names, *dst_ty*'s pointer will also be marked as an invalid type pointer. In the scenarios of (Con → Gen) and (Gen → Con), they follow the same logic to check when the arbitrary types or the limited types in *ty_set* could make the type conversion match the two patterns above.

Mismatched Scope Detector (Type III). In order to find the bug efficiently, we define two type sets based on the scope of values: *weak_ty* and *strict_ty*. *weak_ty* represents the type that has a weak constraint on its bit pattern, such as integer and float type. In contrast, *strict_ty* means a strong limitation on the bit pattern, such as bool, string, and character. If the type is found to be a composite type, which has multiple fields, mismatched scope detector will analyze each field and define it as *strict_ty* if one of the fields is included in *strict_ty*. The type conversion between *weak_ty* and *strict_ty* can create a type with an invalid bit pattern.

Bug Definition. There are two patterns of conversion that can create an invalid type: 1) *src_ty* belongs to *weak_ty* while *dst_ty* belongs to *strict_ty*. 2) *src_ty* and *dst_ty* are in *strict_ty* and *weak_ty*, while *dst_ty*'s pointer is mutable. In these two types of conversions, an invalid type pointer can be created.

Type Conversion. If the detector finds the conversion in (*weak_ty* → *strict_ty*), it will mark the *dst_ty*'s pointer as an invalid type pointer since the bit pattern of *src_ty* could be invalid for *dst_ty*. When the conversion is found in (*strict_ty* → *weak_ty*), mismatched scope detector will take a further analysis on whether the *dst_ty* is mutable since changing the bit pattern of *dst_ty* can also make *src_ty* invalid. In the scenarios of (Con → Gen) and (Gen → Con), the detector follows the same logic to check whether the type conversion is performed between *weak_ty* and *strict_ty* with mutability analysis.

4.2.2 Access Check

Access check is performed to analyze how the invalid type pointer captured by Type conversion check can be accessed. The analysis can be separated into two steps: 1) check whether the pointer is accessed in the function, and 2) analyze whether the pointer is accessible for the caller function. As the first step, to check whether *dst_ty*'s pointer is accessed in the function, we focus on the dereference in statements and the unsafe function calls in the terminators of the MIR. For statements, we check whether *dst_ty*'s pointer is aliased with the dereferenced pointer. For unsafe function calls, we collect a list of unsafe functions that are widely used in the core libraries of Rust, such as `ptr::read/copy`, `ptr::as_ref`, and `slice::from_raw_parts`, which requires the pointer refers to an aligned, consecutively initialized type. The access list for mismatched scope detector also includes other APIs such as `str::from_utf8_unchecked` or `CStr::from_ptr`. These functions require types to be encoded with the specific bit patterns. Access check will verify whether the pointer passed in these unsafe functions is aliased with the *dst_ty*'s pointer.

In the second step, to check whether *dst_ty*'s pointer is accessible for the caller function, we analyze whether the *dst_ty*'s is aliased with the return type only when the return type is a reference. When the return type is a raw pointer, accessing it requires the `unsafe` block since Rust does not guarantee the safety of the raw pointer. Since using `unsafe` highlights the responsibility for the bugs and we only consider the function that performs the problematic type conversion to be the culprit of bugs, we will set up the requirement for the return type to be a reference in the second step. After access check ensures that the pointer of invalid *dst_ty* can be accessed, the bug report will be generated as the output of bug detector.

4.2.3 Developer-Enforced Check Analysis

Developer-Enforced Check is usually used by the developers to prevent type confusion bugs manually. Through examining these checks, we can confirm that the developer has handled the type conversion errors, further reducing the false alarms of TYPEPULSE. We categorize them into two scenarios: *Pre Type Check* and *Post Type Check*, where the check inserted before and after type conversions, respectively.

We summarize various patterns of developer-enforced checks that address type confusion bugs individually. First, the pre type checks used to prevent misalignment bugs include calling `align_of` and `alloc`, which are used to check and assign memory layout before the type conversion. There is also a post type check that the developer uses to safely load the misaligned type e.g., `read_unaligned`. Second, for the inconsistent layout bug, pre type check is used to guarantee the memory is completely initialized. The typical patterns include using `size_of` to restrict the size at run-time. For example, if the struct contains two fields of `u32` types, developers can

check if the size of the struct type is 8 bytes, further ensuring that no padding bytes are inserted. The post type checks such as `ptr::write`, which can be used to access the uninitialized memory, will also be detected before TYPEPULSE raises the alarms for the bugs. Detecting developer-enforced checks for scope mismatch bugs is challenging because developers often use runtime value comparisons.

4.2.4 Integration of Interprocedural Analysis

Interprocedural analysis plays an essential role in confirming the presence of the type confusion bug. In this section, we describe how it is incorporated into bug detector.

Type Conversion Check. We can use it to identify type conversions between functions. For instance, consider (Con → Con), where we notice an unsafe type casting from a `u8` pointer to a `u16`. According to our misalignment bug criteria, this should trigger an alert. Nevertheless, through interprocedural analysis, we identify a type conversion from a `u16` pointer back to `u8` in the caller function. As a result, the `src_ty` should be `u16` and properly aligned to two bytes, which means that there is no any misalignment. Considering another case with generic type (Gen → Con) and type conversion being detected in a method, we leverage property graph to identify the type constructor function and analyze the type conversion pairs in the method.

Access Check. Given that the type may not be accessible within the current function, we also examine the callee functions, such as a raw pointer dereference. Additionally, we gather certain `unsafe` standard library functions, which involve type access, accelerating the verification of type access.

Developer-Enforced Check Analysis. Developer-Enforced Check could also exist in external functions. In other words, it can be implemented in callers, type constructors, or callees. Thus, TYPEPULSE must analyze all reachable functions to locate the related type checks.

4.3 Implementation

TYPEPULSE is developed with 5249 lines of code in Rust, utilizing *rustc* and fully integrating with *Cargo*, Rust's official package manager. TYPEPULSE focuses on target files that can be compiled into an executable or a library [48]. Using *Cargo*, we address dependency issues prior to compilation and identify all targets in the package suitable for analysis. Compilation of these target files is done through *rustc*. Upon completion, TYPEPULSE is activated within the `after_analysis` callback function of the *rustc* driver, which is triggered by *rustc* following the generation of Rust compiler's MIR, allowing us to employ the resulting MIR data as input for property graph constructor to start the analysis.

The workflow of TYPEPULSE can be divided into two phases: 1) detecting if the type conversion generates a problematic `dst_ty`, and 2) checking if the problematic type is

Table 3: Bugs identified by TYPEPULSE; note that we have only listed the bugs that have been reported for over three months as of January 1, 2025.

Package	Version	Stars	Bug Types‡ and Numbers	Status*	Patched†
candle-core	0.4.1	13.2k	Con → Gen: I:3 (as)	●	-
py-spy	0.3.14	11k	Con → Con: I:1 (as)	○	-
fyrox-core	0.27.0	7.1k	Con → Con: I:1 (as) Gen → Con: II:4 (as), III:2 (as)	●	✓
gfx-backend-gl	0.9.0	5.2k	Con → Gen: I:1 (as)	●	-
silicon	0.5.2	3.1k	Con → Con: II:1 (as)	○	-
webrender	0.61.0	3k	Con → Con: I:2 (as)	●	-
spl-token-swap	3.0.0	2.3k	Con → Gen: I:1 (as), III:1 (as)	●	-
scryer-prolog	0.9.4	1.9k	Con → Con: I:6 (transmute)	●	-
libafl	0.10.1	1.6k	Con → Con: I:3 (as), III:4 (as)	●	✓
mesalink	1.1.0	1.5k	Gen → Con: II:1 (as)	○	-
fontdue	0.8.0	1.2k	Gen → Con: I:1 (transmute)	●	-
pprof	0.13.0	1k	Con → Con: II:1 (as) Con → Con: I:1 (as)	●	✓
rendy-core	0.5.1	814	Gen → Con: II:2 (as)	◐	-
rendy-util	0.4.1	814	Gen → Con: II:2 (as)	◐	-
sciter-rs	0.5.58	784	Gen → Con: I:1 (as)	●	-
rosrust	0.9.11	728	Gen → Con: II:3 (as) III:1 (as)	◐	-
cortex-m	0.7.7	669	Gen → Con: II:1 (as)	●	✓
rafx-base	0.0.15	574	Gen → Con: II:2 (as)	◐	-
xous	0.9.50	500	Con → Gen: I:2 (as), III:2 (as)	●	✓

* Bug status: ○: Reported; ●: Confirmed by Vendors; ◐: Verified by PoC.
† Patch status: ✓: already patched; -: unpatched.
‡ Bug type: Con: concrete type; Gen: generic type; [I, II, III]: bug type.

accessed. With pairs of type sets generated by type conversion analysis, we can find a problematic type conversion even when a generic type is involved. With the alias graph built by pointer alias analysis, we can track how the pointer can be accessed. The type conversion pairs and alias graph are stored in property graph, which accelerates the interprocedural analysis to obtain the information of external functions. For interprocedural analysis, we introduced a depth limitation to avoid the path explosion problem. We set the path length to 1 (tracing only the immediate caller or callee function), aligning with that in Rupta [31].

5 Evaluation

Dataset Collection. We gathered packages from `crates.io`, the Rust community's crate registry. To ensure comprehensive bug detection, The dataset consists of the packages ranked by download counts and the number of GitHub stars (as of September 1, 2024), and each of them has more than 500 stars. Regarding package size, the largest package contains 510k LoC, with an average package size of 9k LoC.

Experiment Setup. We built TYPEPULSE and conducted experiments on a server with 48-core Intel Xeon CPU ES-2630 and 256 GB memory. The server was deployed with Ubuntu 22.04 and `rustc` 1.72.0-nightly. For each package, we set the preparation (dependencies resolution and compilation) time threshold to 20 minutes and TYPEPULSE detection time threshold to 2 minutes. We ran TYPEPULSE on the 3,000 packages for detection.
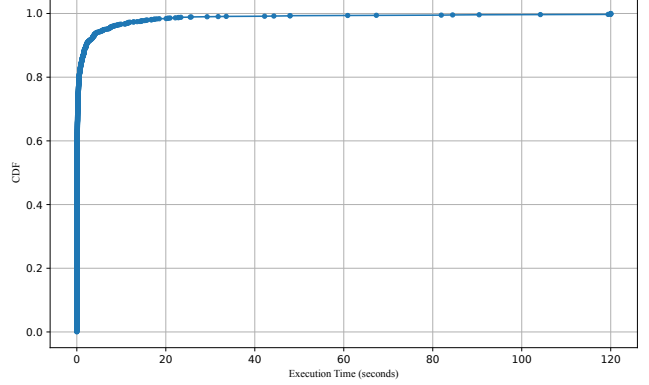

Fig. 2: Detection time of TYPEPULSE.

5.1 Bug Detection Results

First, we ran TYPEPULSE on the RUSTSEC dataset of existing type confusion bugs as of September 1, 2024 (see Table 1). TYPEPULSE is capable of detecting all existing type confusion bugs of three types. Second, we ran TYPEPULSE with 16-thread to scan the 3,000 packages. It ran 18 hours in total; the majority of time was spent on resolving dependencies and compilation. The detection time only took about 80 mins and 94 bugs were reported. For the 3,000 packages, 618 (20.6%) failed due to compiler version, 37 (1.2%) did not have proper cargo metadata, and 309 (10.3%) failed with custom build options, which cannot be resolved automatically and required manual work. All the remaining 2,036 packages (67.9%) could successfully compile within 20 minutes. Among them, 282 did not have /bin or /lib targets for detection, so TypePulse did not execute on them. For 1,754 packages that TypePulse ran detection on, the average detection time was 1.71s. Among the 1,754 packages, 1,483 were completed within 1s; 1,652 were completed within 5s; only 59 needed more than 10s to complete; 5 packages did not finish execution within 2 minutes. The CDF figure of the detection time is shown in Fig. 2. We manually verified these bug reports and confirmed that 71 of them were true positive, indicating an overall precision of 75.5%. The information on packages and bugs that have either been confirmed or reported for over three months is provided in Table 3. Many bugs we detected are not trivial. Among the 71 type confusion bugs detected by TYPEPULSE, there are 50 bugs found in packages with more than 1,000 stars on GitHub. These packages are well known in the Rust community and are usually maintained well by professional teams. For example, `candle-core` [25] is developed by a famous AI company (Hugging Face) [24]. Therefore, our results suggest that even domain experts may write error-prone code in Rust.

Capability to Find New Bugs. To facilitate the resolution of these bugs, we also created a Proof of Concept (PoC) to report

Table 4: The results of detectors on top 3,000 packages.

Detector	Metrics	Bug Types			Overall
		I	II	III	
TypePulse	TP	32	24	15	71
	FP	6	4	13	23
	Precision	84.2%	85.7%	53.6%	75.5%
TypePulse w/o IPA	TP	32	24	13	69
	FP	11	6	13	30
	Precision	74.4%	80%	50%	69.7%

IPA: Interprocedural Analysis.

```
1  impl<const N: usize> String<N> {
2      pub fn new() -> String<N> {
3          String { bytes: [0; N], len: 0 }
4      }
5      pub fn to_str(&self) -> &str {
6          unsafe {
7              str::from_utf8_unchecked(&self.bytes[0..]) }
8      }
9      pub fn push(&mut self, ch: char) -> Result<..> {
10         match ch.len_utf8() {
11             1 => { ... }
12             _ => {
13                 let mut bytes: usize = 0;
14                 let mut data: [u8; 4] = [0; 4];
15                 let subslice = ch.encode_utf8(&mut data);
16                 // ...
```

Listing 5: The false positive case of Mismatched Scope bug.

identified issues to package maintainers, which requires locating undefined behaviors in packages based on the diagnostic information produced by TYPEPULSE. To trigger potential bugs, we need to create the appropriate *src_ty* and *dst_ty* before invoking the problematic conversion. We consider two distinct scenarios. First, when the source and destination variables are openly accessible, such as through public members within a class, we can directly create these variables. Second, when the source and destination variables are private and not directly accessible, a constructor is required to initialize the variables. Additionally, TYPEPULSE helps find feasible constructor functions, which also accelerates the PoC generation.

At the time of writing, we have documented all the issues (71) that we have manually confirmed, and 32 of these have received acknowledgment responses from the package maintainers. After notifying the maintainers, we further reported these issues to the RUSTSEC advisories and CVE database. So far, we have received six RUSTSEC IDs and one CVE ID, and are awaiting confirmation and releases for the remaining ones.

5.2 False Positive Analysis

According to Table 4, there are 23 instances of false positives. There are 2 arising from misidentifying function visibilities and 3 from developers' tricks to prevent invalid pointer exposure. Here we investigate the remaining 18 cases that arise from misinterpretation during the developer-enforced check analysis. It shows that TYPEPULSE cannot understand complex condition semantics in checks, especially when the check is implemented in an unusual way. Considering Listing 5 for example, TYPEPULSE considers the method to_str could construct an illegal string from an array of u8, which might contain a non-utf8 character. After manually studying the type constructor function (new) and the method used to insert characters into the string (push), we find that the characters in the string are restricted to be utf8 by the encoding at line 15 (encode_utf8). This complex encoding makes it difficult for TYPEPULSE to avoid false alarms in detecting Mismatched Scope Bugs. So far, TYPEPULSE can only detect standard developer-enforced check patterns and several unsafe APIs with clear safety documentation. It is challenging for TYPEPULSE to understand such encoding operations.

We will extend the capability of TYPEPULSE to interpret developer-enforced check in future work.

5.3 Impacts of Interprocedural Analysis

We conduct an ablation study to demonstrate TYPEPULSE's capabilities in reducing false positives by disabling interprocedural analysis (refer to the second row in Table 4). TYPEPULSE reduces 6 more false positives and detects 2 more true positives with interprocedural analysis enabled. Scanning 3k packages, TYPEPULSE obtains the results with precision 75.5%, consisting of 71 True Positives and 23 False Positives. After disabling the interprocedural analysis, TYPEPULSE's precision is reduced to only 69.7%, consisting of 69 True Positives and 30 False Positives. This comparison result highlights that interprocedural analysis can significantly enhance precision by analyzing the context across functions. First, it can detect the positive cases relying on external functions to decide the bit patterns – 2 more mismatched scope bugs were detected. Second, it can reduce the false alarms in cases with developer-enforced checks – 7 more False Positives were reduced. For example, in the package of arrow-buffer, disabling interprocedural analysis will cause four more false positive cases of misalignment bugs. The Listing 6 shows a false positive case mitigated by interprocedural analysis. TYPEPULSE first locates suspicious type conversion at line 20 since it finds that the pointer of buffer is cast to arbitrary generic type. However, the generic type cannot be controlled by attackers since as_slice is a method relying on BufferBuilder and casting MutableBuffer to generic type. The constructor function of MutableBuffer cannot be located by traditional interprocedural analysis because it is not the caller of the method. TYPEPULSE implements the functionality to find out the constructor functions by matching the type to the ones returned from other functions. In this example, TYPEPULSE locates the function with_capacity, which returns MutableBuffer as the constructor function. After an-

```
1  impl MutableBuffer {
2      #[inline]
3      pub fn with_capacity(capacity: usize) -> Self {
4          let layout = Layout::
5              from_size_align(capacity, ALIGNMENT).unwrap();
6          let data = match layout.size() {
7              0 => dangling_ptr(),
8              _ => {
9                  let raw_ptr = unsafe {
10                     std::alloc::alloc(layout)
11                 };
12         // ...
13         Self { data, len: 0, layout }
14 // ...
15 impl<T: ArrowNativeType> BufferBuilder<T> {
16     #[inline]
17     pub fn as_slice(&self) -> &[T] {
18         // ...
19         unsafe { std::slice::from_raw_parts(
20             self.buffer.as_ptr() as _, self.len) }
21 // ...
```

Listing 6: The false positive case resolved by interprocedural analysis.

alyzing the constructor function, TYPEPULSE finds that the constructor function already guarantees the alignment of type with Layout::from_size_align; therefore, this should be a false alarm. Without interprocedural analysis, TYPEPULSE cannot detect the developer-enforced check implemented in the constructor function.

5.4 Comparison with Existing Tools

To the best of our knowledge, TYPEPULSE is the first bug detection tool to systematically detect type confusion bugs in Rust, so we are unable to find similar tools to perform an *apple-to-apple* comparison. we choose the tools that are able to partially detect type confusion bugs for comparison: Clippy [9] and Rudra [15]. We run them on the packages listed in Table 3 and compare the results. The comparison results are shown in Table 5. Additionally, we compare the performance of TYPEPULSE to the Rust's type system on the existing type confusion bugs in RUSTSEC dataset (see Table 1).

Comparison with Clippy. Clippy is a static analysis tool that implements more than 650 types of lints [67] to detect common errors in the Rust program. Clippy supports 2 types of lints to find the unsound usages of as and transmute. First, it can check whether as can lead to a misaligned pointer (cast_ptr_alignment [1]). Second, it can check whether transmute occurs between types of different Application Binary Interfaces (ABIs) (unsound_collection_transmute [2]). The version of Clippy with which we compare is 0.1.72, and Table 5 reveals that Clippy identifies only 10 relevant bugs, all of which

Table 5: Comparison with Clippy and Rudra.

Detector	Type I	Type II	Type III	Overall
Clippy	10	0	0	10
Rudra	0	0	0	0
TYPEPULSE	32	24	15	71

are misalignment issues (Type I) restricted to $Con \rightarrow Con$ (21 bugs in total). Regarding the remaining 11 misalignment bugs, one warning is intentionally suppressed by developers, four arise from overlooking variations in ABIs, and six involve transmute. We summarize two main reasons for Clippy's limitation as follows. Firstly, Clippy fails to identify potential bugs involving generic types, as the two lint checks are only carried out when both the *source type* and *dest type* are concrete. Secondly, Clippy does not have a comprehensive approach to identifying type conversion errors. It is observed that Clippy's checks vary between as and transmute; misalignment checks are applied to as but omitted for transmute. For inconsistent layout bugs, checks are conducted exclusively on transmute. Furthermore, upon reviewing developer comments, it is observed that developers often disregard the warnings due to their belief that Clippy generates a significant number of false positive cases.

Comparison with Rudra. Rudra [15] is the bug detector that can be used to capture memory safety bugs from Rust packages. We compared TYPEPULSE with Rudra for two reasons. First, Rudra is the state-of-the-art memory safety bug detector, reporting 51.6% of all memory safety bugs. Second, Rudra claims it can find the bugs of uninitialized memory exposure, which is the result of inconsistent layout bugs. Our evaluation results show that Rudra can find five bugs of uninitialized memory. However, none of them occurs in the type conversion process, which means it is not effective in detecting type confusion bugs. The main reason is that, for the patterns of type conversion, Rudra only detects the terminators at the MIR level. Unfortunately, transmute has been translated to statements rather than terminators since 2021. Moreover, Rudra implements the dataflow checker of transmute but excludes as. As a result, Rudra is not able to detect type confusion bugs effectively.

Comparison with Rust Type System. We also conduct the experiment to elaborate if the existing Rust type system can effectively detect type confusion bugs in unsafe code regions. We compare the positive cases of type confusion bugs detected by TYPEPULSE and the Rust type system. Since Rust considers unsafe keyword to be required for raw pointer dereferences and unsafe APIs, removing unsafe will introduce syntax errors. For the purpose of calculating the precision, we count these unremovable unsafe as positive bug detections by Rust type system. If there are multiple unsafe blocks in a single function, we count it as one. In other words, the functions with unsafe but no type confusion bugs being

```
1   impl<'a, T> Iterator for TempFdArrayIterator<'a, T> {
2       type Item = &'a T;
3       fn next(&mut self) -> Option<Self::Item> {
4           // ...
5           let length = self.file_vec.len() / size_of::<T>();
6           let ts = unsafe {
7               slice::from_raw_parts(
8                   self.file_vec.as_ptr() as *const T,
9                   length)
10              };
11
12  pub fn build(&self) -> Result<Report> {
13          let mut hash_map = HashMap::new();
14          match self.profiler.write().as_mut() {
15              Err(err) => {...}
16              Ok(profiler) => {
17                  profiler.data.try_iter()?
18                      .for_each(|entry| { .. }
```

Listing 7: Misaligned bug found in the `pprof` package.

reported should be considered as false positive cases. We use all vulnerable files including the 32 existing bugs (Table 1) as the benchmark and find that both TYPEPULSE and the type system can detect all the existing type confusion bugs; however, the type system produces 116 false positive cases while TYPEPULSE only produces 3.

Summary. Our evaluation confirms that TYPEPULSE is the most effective tool to detect type confusion bugs in Rust. Existing state-of-the-art Rust bug detection tools such as Clippy and Rudra are not as effective, since they are not designed for detecting type confusion bugs. Our experiment also explains that the current Rust type system is not sufficient to check the unsafe type conversion, highlighting the necessity of TYPEPULSE.

5.5 Impacts of Type Confusion Bugs

Memory errors might occur due to type confusion bugs, especially if the target type allows access to memory that the source type cannot reach. The type confusion bugs discovered by TYPEPULSE have different security implications. Among them, 28 trigger panics, 24 cause uninitialized memory access, 8 lead to out-of-bounds access, 7 construct illegal types, and 4 can generate data race issues. Using case studies of the `pprof` package [7], a popular Rust-based CPU profiler with 1.3k stars on Github and 159 crates.io dependents, we show the impacts of bugs and how it helps diagnose Rust performance bottlenecks. The bug was reported and confirmed by the developers on Github. Besides, we also provide other case studies and corresponding PoCs in Appendix A.

Misalignment Bugs. As shown in Listing 7, TYPEPULSE detects a misalignment bug (line 8) in the `next` function implemented on `TempFdArrayIterator`. When the unsafe `slice::from_raw_parts` is called (line 11), it assumes the caller meets safety contracts. The raw pointer must be aligned, non-null, and point to `length` bytes of initialized values [11]. Violating these safety contracts causes a panic. One way to invoke the `next` function is to build reports from a running profiler; it will iterate each entry to process the data and write them into reports (line 12). The generic type `T` in `TempFdArrayIterator` is decided by the item stored in the `profiler.data` (line 17), which is `UnresolvedFrames`. `UnresolvedFrames` is a representation of an event backtrace, and it is a self-defined `struct` type with fields of u64, usize, an array of u8. Since the `file_vec` is aligned to 1 byte, any type that has a larger alignment than 1 byte can lead to the misalignment bug and crash here.

Famous Rust-based applications like GreptimeDB [5] (4.1k stars on GitHub) are affected by this bug. GreptimeDB, a time series database storing logs, events, and CPU usage, crashes when calling pprof (`report::ReportBuilder::build`) to build reports. This panic can obstruct queries or data writes, causing real-time network monitoring applications to miss identifying high CPU usage, leading to network performance decline. For example, given a GreptimeDB-based public network server that provides the interface of event backtrace, the attacker could initialize the data `T` with the type aligned to 2 bytes, causing the network server panic and rendering it unavailable to users. After reviewing related GitHub issues, we traced the `ReportBuilder::build` code pattern and searched on GitHub. Over 230 code files show similar patterns and may be affected.

6 Discussion

Rust vs. C++ on Type Confusion Bugs. The distinction of compiler features, type systems, and memory management in Rust and C++ lead to different types of type confusion bugs and new challenges in detecting them. First, while pointer-type conversion is the primary cause of bugs in both languages, the inconsistent layout bug and the mismatched scope bug mentioned in this paper do not occur in C++. This is because the C++ compiler does not rearrange the memory layout of composite types by default (thus no *unstable_ty*), and it also does not impose strict requirements on the bit-pattern as Rust does. From a different angle, type confusion bugs in C++, which consistently occur when downcasting an object from a parent class to a child class, does not exist in Rust. This is because Rust lacks the object characteristics needed for such issues. Rust introduces the concept of a trait object [45], comparable to C++ objects, to define shared behaviors. However, since trait objects do not involve inheritance, they prevent object-based type confusion bugs. Second, before resolving generic types, extracting traits completely is more difficult in Rust since Rust's traits can be implicitly bounded while C++'s concept must be explicit. Lastly, Rust's implicit memory management brings convenience to developers by avoiding manual management. However, it actually makes pointer alias

analysis harder. To verify whether the pointer is still valid, the detector must ensure whether the memory is automatically dropped or the ownership is transferred.

Complementing Rust type system. TYPEPULSE is necessary since expanding the type system protection from Safe Rust to include `unsafe` is insufficient, as shown in §5.4. The type system's safety assurances are not derived from executing type checks. Rather, Safe Rust prevents the presence of an *invalid type* right from the beginning. Consider the example of a misaligned pointer: Safe Rust prevents developers from converting a pointer to one aligned with larger byte sizes (refer to Listing 1). Due to this stringent rule, the compiler confidently assumes that misaligned pointers are not present in Safe Rust. Consequently, optimizations and code generation within the compiler's backend are carried out based on this assumption. A pointer generated within `unsafe` is the only entity capable of circumventing these rules. Even if the pointer from `unsafe` is converted to a reference, the compiler will still treat this reference as reliable. This flawed assumption can result in undefined behavior. Therefore, a tool like TYPEPULSE that aids in identifying an invalid type becomes essential for Rust developers.

Mitigations of Type Confusion Bugs. We summarize common ways to fix the bugs detected by TYPEPULSE.

Type I: misalignment bugs. Misaligned references are prohibited in safe code, but the safety of misaligned raw pointers depends on access methods. We propose two strategies to prevent misalignment issues before dereferencing: (1) use `read_unaligned` [8] or `write_unaligned` [14], which handle misaligned pointers, or (2) create a new aligned pointer. Most functions require aligned pointers; `read_unaligned` creates an aligned duplicate by copying data (`copy_nonoverlapping`) and casting to `u8`. Alternatively, developers can manually create a new pointer by adding an offset (`ptr.add`) to align the address.

Type II: inconsistent layout bugs. To avoid the inconsistent layout bug, we need to ensure the type's memory layout is consistent and stable. If *dst_ty* is a primitive type with initialized memory in consecutive bytes, *src_ty* must not have uninitialized bytes. Although most code avoids inconsistent type conversion, it can occur during generic type conversion. To prevent bugs in generic types, we can apply trait bounds — list the types that can be legally converted and implement the trait on them. This ensures callers use only the defined types as parameters. A well-known trait implementing this concept is `bytemuck::Pod` [49]. For `struct`-to-`struct` conversions, developers often wrongly assume stable memory layout. To ensure stability, developers need to annotate types for conversion with `repr(C)` or `repr(transparent)` [46].

Type III: mismatched scope bugs. Such bugs often occur in exposed APIs with generic type conversion. Developers should limit types and use trait bounds to restrict conversion and validate values before converting. Li-braries provide `unsafe` APIs like `from_*_unchecked` (e.g., `str::from_utf8_unchecked` [61]) for type conversion, whose safety must be ensured by callers. Callers must validate that source type values are appropriate for destination types. For instance, `str::from_utf8_unchecked` requires UTF-8 valid input, unlike the safe `std::from_utf8` [60], which checks this. Developers often use `from_utf8_unchecked` over `from_utf8` to avoid overhead, but safe functions should be used in critical security scenarios.

7 Limitations and Future Work

As detailed in §5.2, TYPEPULSE has limitations in accurately interpreting different implementations of developer-enforced checks, leading to false positive cases. Some check patterns are inherently implicit and difficult to formalize. For example, calling size check functions but actually examining padding bytes. Understanding this intricacy requires a deep contextual insights. Additionally, assessing the value within the check condition is particularly challenging when relying solely on static tools. Without incorporating dynamic analysis, assuring the accuracy of security checks becomes difficult, especially in large software projects. In future work, we aim to integrate symbolic execution in access checks, such as implementing constraints to confirm that a pointer's memory address cannot be a multiple of the type size before TYPEPULSE flags misalignment issues. Furthermore, with a path limitation of 1 for interprocedural analysis, TypePulse struggles to grasp the context in extended call chains, which we aim to address in future work.

8 Related Work

Research on type confusion bugs in C++/Javascript. In other programming languages, there are a large number of existing works focusing on type confusion bugs. For instance, C++ supports implicit type conversion which can lead to significant issues; thus, numerous scholars have developed various methods to identify the type confusion bugs [18, 21, 26, 29]. Given that C++ includes diverse type casting abilities and runtime polymorphism, detectors for such bugs must integrate runtime analysis techniques while also managing performance overhead. To enhance performance, `TypeSan` [21] developed a framework capable of efficiently monitoring memory allocation details. There are also several research works focusing on type confusion bugs in Javascript [16, 41, 59]. Type-related issues in C++ are deemed more severe as they directly contribute to memory safety problems, whereas Javascript typically operates within constrained settings like web browsers. Although Rust is engineered with improved type-safety compared to C++ and Javascript, our research shows that type-related errors can still occur in Rust.

Research on unsafe Rust. A substantial body of research explores how the use of unsafe can compromise the integrity of Rust programs [19,27,34,39,44,47,68,69]. Xu et al. analyzed hundreds of memory-safety issues, determining that safety assurances can be violated by unsafe code [68], while other works study how to protect the Rust program [27,47]. Additionally, some scholars have investigated both memory-safety and concurrency issues, assessing the effects of eliminating unsafe code [44]. Rudra [15] and MirChecker [33] are specifically designed to target functions that incorporate unsafe code and detect memory-safety issues. Observations from our research also suggest a significant correlation between type confusion bugs in Rust and the use of unsafe code.

9 Conclusion

In this paper, we develop TYPEPULSE, the *first* static analysis tool for detecting type confusion bugs in Rust. TYPEPULSE focuses on detecting the three most common categories of type confusion bugs— misalignment, inconsistent layout, and mismatched scope. TYPEPULSE detected 71 previously unknown bugs from the top 3,000 Rust packages. This number surpasses the number of type confusion bugs documented in the last five years in RustSec, which shows the effectiveness of TYPEPULSE. The identified bugs were reported to the developers, who have confirmed 32 of these issues. We also compare TYPEPULSE with existing Rust bug detection tools, and perform case studies to demonstrate the security implications of the identified bugs. TYPEPULSE will be open-sourced to facilitate future research.

Acknowledgment

Open Science

To promote transparency and reproducibility in our research, the data artifacts of this paper will be made publicly available, including source code, detected bugs, and related github issues we reported. We disclose only those issues that have been acknowledged and resolved by developers. Issues that remain unresolved at the time of writing are not included in detail. All data is available on Zenodo: https://zenodo.org/records/14750104.

Ethics Considerations

We take ethics seriously in this project. All Rust repositories we tested in the paper are publicly accessible on Github. During evaluations of our work, TYPEPULSE identified several previously unknown type confusion vulnerabilities in widely used software. In each case, we followed a responsible disclosure policy, and reported our discovered vulnerabilities to the developers. We also submitted our findings to the CVE program and the RustSec Advisory Database. We did not disclose those issues to anyone else. All the examples mentioned in the paper are the issues that have been acknowledged and fixed. The RustSec IDs issued are: RUSTSEC-2023-0046, RUSTSEC-2023-0047, RUSTSEC-2024-0408, RUSTSEC-2024-0424, RUSTSEC-2024-0426, RUSTSEC-2024-0431; The CVE ID is currently in the reserved status at the time of writing and will be released later.

References

[1] Clippy lints - cast pointer alignment. https://rust-lang.github.io/rust-clippy/master/index.html#/cast_ptr_alignment.

[2] Clippy lints - unsound collection transmute. https://rust-lang.github.io/rust-clippy/master/index.html#/unsound_collection_transmute.

[3] Foreign function interface - the rustonomicon. https://doc.rust-lang.org/nomicon/ffi.html.

[4] Generic data types - the rust programming language. https://doc.rust-lang.org/book/ch10-01-syntax.html.

[5] Greptime. https://greptime.com/.

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

[7] pprof. https://crates.io/crates/pprof.

[8] read_unaligned in std::ptr - rust. https://doc.rust-lang.org/nightly/std/ptr/fn.read_unaligned.html.

[9] rust-lang/rust-clippy: A bunch of lints to catch common mistakes and improve your rust code. book: https://doc.rust-lang.org/clippy/. https://github.com/rust-lang/rust-clippy/tree/master.

[10] Rustsec: The rust security advisory database. https://rustsec.org/advisories/.

[11] std::slice::from_raw_parts. https://doc.rust-lang.org/std/slice/fn.from_raw_parts.html.

[12] Trait and lifetime bounds - the rust reference. https://doc.rust-lang.org/reference/trait-bounds.html.

[13] Typeid in std::any - rust. `https://doc.rust-lang.org/stable/std/any/struct.TypeId.html`.

[14] write_unaligned in std::ptr - rust. `https://doc.rust-lang.org/stable/std/ptr/fn.write_unaligned.html`.

[15] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. Rudra: Finding memory safety bugs in rust at the ecosystem scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 84–99, New York, NY, USA, 2021. Association for Computing Machinery.

[16] Fraser Brown, Shravan Narayan, Riad S. Wahby, Dawson Engler, Ranjit Jhala, and Deian Stefan. Finding and preventing bugs in javascript bindings. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 559–578, 2017.

[17] Mohan Cui, Chengjun Chen, Hui Xu, and Yangfan Zhou. Safedrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis. *ACM Trans. Softw. Eng. Methodol.*, 32(4), may 2023.

[18] Gregory J Duck and Roland HC Yap. Effectivesan: type and memory error detection using dynamically typed c/c++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–195, 2018.

[19] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. Is rust used safely by software developers? *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 246–257, 2020.

[20] Xiaokang Fan, Zeyu Xia, Sifan Long, Chun Huang, and Canqun Yang. Accelerating type confusion detection with pointer analysis. *IAENG International Journal of Computer Science*, 20:664–671, 2020.

[21] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik Van Der Kouwe. Typesan: Practical type confusion detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 517–528, 2016.

[22] Valerii Hiora. lmdb-rs: Rust bindings for lmdb. `https://crates.io/crates/lmdb-rs`.

[23] The White House. Future software should be memory safe. `https://www.whitehouse.gov/oncd/briefing-room/2024/02/26/press-release-technical-report/`.

[24] The HuggingFace. The ai community building the future. `https://huggingface.co/`.

[25] The HuggingFace. Candle - minimalist ml framework for rust. `https://github.com/huggingface/candle`.

[26] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. Hextype: Efficient detection of type confusion errors for c++. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2373–2387, 2017.

[27] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. Pkru-safe: Automatically locking down the heap between safe and unsafe languages. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 132–148, 2022.

[28] The Rust Programming Language. What is ownership? `https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html`.

[29] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. Type casting verification: Stopping an emerging attack vector. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 81–96, 2015.

[30] Tuo Li, Jia-Ju Bai, Yulei Sui, and Shi-Min Hu. Path-sensitive and alias-aware typestate analysis for detecting os bugs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 859–872, New York, NY, USA, 2022. Association for Computing Machinery.

[31] Wei Li, Dongjie He, Yujiang Gui, Wenguang Chen, and Jingling Xue. A context-sensitive pointer analysis framework for rust and its application to call graph construction. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*, CC 2024, page 60–72, New York, NY, USA, 2024. Association for Computing Machinery.

[32] Xuejian Li and Zhengguang Zhu. Software defect detection based on feature fusion and alias analysis. In *2023 IEEE International Test Conference in Asia (ITC-Asia)*, pages 1–6, 2023.

[33] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C.S. Lui. Mirchecker: Detecting bugs in rust programs via static analysis. CCS '21, page 2183–2196, New York, NY, USA, 2021. Association for Computing Machinery.

[34] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. Cross-language attacks. In *NDSS*, 2022.

[35] Mozilla. Mozilla firefox. `https://www.mozilla.org/en-US/firefox/new/`.

[36] NIST. Nvd - cve-2023-3079. https://nvd.nist.g ov/vuln/detail/CVE-2023-3079.

[37] NIST. Nvd - cve-2023-4762. https://nvd.nist.g ov/vuln/detail/CVE-2023-4762.

[38] NIST. Nvd - cve-2024-1939. https://nvd.nist.g ov/vuln/detail/CVE-2024-1939.

[39] Michalis Papaevripides and Elias Athanasopoulos. Exploiting mixed binaries. *ACM Transactions on Privacy and Security (TOPS)*, 24(2):1–29, 2021.

[40] Plain. Trait plain - doc.rs. https://docs.rs/plai n/latest/plain/trait.Plain.html.

[41] Michael Pradel, Parker Schuh, and Koushik Sen. Typedevil: Dynamic type inconsistency analysis for javascript. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 314–324. IEEE, 2015.

[42] The Rust Project. Rust Programming Language. http s://www.rust-lang.org/.

[43] The Rust Project. Type conversions - the rustonomicon. https://doc.rust-lang.org/nomicon/conv ersions.html.

[44] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. Replication package for article: Understanding memory and thread safety practices and issues in real-world rust programs. *Artifact Digital Object Group*, 2020.

[45] The Rust Reference. Trait objects. https: //doc.rust-lang.org/reference/types/trai t-object.html.

[46] The Rust Reference. Type layout. https://doc.ru st-lang.org/reference/type-layout.html.

[47] Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow. Keeping safe rust safe with galeed. In *Proceedings of the 37th Annual Computer Security Applications Conference*, pages 824–836, 2021.

[48] Rust. Cargo targets - the cargo book. https://doc.rust-lang.org/cargo/refere nce/cargo-targets.html.

[49] Rust. Pod in bytemuck. https://docs.rs/bytemu ck/latest/bytemuck/trait.Pod.html.

[50] Rust. Traits. https://doc.rust-lang.org/refe rence/items/traits.html.

[51] rustc_middle. rustc_middle::mir::local. https://doc.rust-lang.org/nightly/nightl y-rustc/rustc_middle/mir/struct.Local.html.

[52] rustc_middle. rustc_middle::mir::statement. https: //doc.rust-lang.org/nightly/nightly-rustc /rustc_middle/mir/struct.Statement.html.

[53] rustc_middle. rustc_middle::mir::terminator::terminator. https://doc.rust-lang.org/nightly/nightl y-rustc/rustc_middle/mir/terminator/struct. Terminator.html.

[54] RUSTSEC. Rustsec-2019-0035. https://rustsec. org/advisories/RUSTSEC-2019-0035.html.

[55] RUSTSEC. Rustsec-2020-0029. https://rustsec. org/advisories/RUSTSEC-2020-0029.html.

[56] RUSTSEC. Rustsec-2022-0074. https://rustsec. org/advisories/RUSTSEC-2022-0074.html.

[57] RUSTSEC. Rustsec-2024-0016. https://rustsec. org/advisories/RUSTSEC-2024-0016.html.

[58] RUSTSEC. Rustsec-2024-0338. https://rustsec. org/advisories/RUSTSEC-2024-0338.html.

[59] Lili Sun, Chenggang Wu, Zhe Wang, Yan Kang, and Bowen Tang. Kop-fuzzer: A key-operation-based fuzzer for type confusion bugs in javascript engines. In *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 757–766, 2022.

[60] The Rust Team. from_utf8 in std::str - Rust. https://doc.rust-lang.org/stable/std/str/f n.from_utf8.html.

[61] The Rust Team. Function from_utf8_unchecked in std::str - Rust. https://doc.rust-lang.org/stab le/std/str/fn.from_utf8_unchecked.html.

[62] The Rust Team. Meet safe and unsafe. https://doc.rust-lang.org/nomicon/meet -safe-and-unsafe.html#meet-safe-and-unsafe.

[63] The Rust Team. unsafe - rust. https://doc.rust-l ang.org/std/keyword.unsafe.html.

[64] The Rust Reference. Behavior considered undefined. https://doc.rust-lang.org/reference/ behavior-considered-undefined.html.

[65] Linus Torvalds and thousands of contributors. The linux kernel. https://www.kernel.org/.

[66] Wikipedia. Breadth-first search. https://en.wikip edia.org/wiki/Breadth-first_search.

```
1  // lmdb-rs/src/traits.rs
2  impl FromMdbValue for $t {
3      fn from_mdb_value(value: &MdbValue) -> $t {
4          unsafe { *transmute(value.get_ref()) }
5      }
6  }
7  // lmdb-rs/src/core.rs
8  #[inline]
9  pub fn new_from_sized<T>(data: &'a T) -> MdbValue<'a> {
10     unsafe { MdbValue::new(transmute(data), size_of::<T>())}
11 }
```

Listing 8: The vulnerabilities in `lmdb-rs` package [22].

```
1  fn main() {
2      let a: i32 = 3;
3      let mdbval = MdbValue::new_from_sized(&a);
4      let res = i64::from_mdb_value(&mdbval);
5      println!("{:?}", res);
6  }
```

Listing 9: Exploit that trigger bug type I.

[67] Wikipedia. Lint (software). https://en.wikipedia.org/wiki/Lint_(software).

[68] Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael R. Lyu. Memory-safety challenge considered solved? an in-depth study with all rust cves. *ACM Trans. Softw. Eng. Methodol.*, 31:3:1–3:25, 2020.

[69] Yuchen Zhang, Yunhang Zhang, Georgios Portokalidis, and Jun Xu. Towards understanding the runtime performance of rust. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–6, 2022.

A Bugs Due to Generic Type Conversion

We demonstrate these three types of type conversion bugs using an example. The bugs are discovered from `lmdb-rs` package [22], which is a package providing API bindings to the LMDB (Lightning Memory-Mapped Database) library. In Listing 8, we showcase an implementation of `from_mdb_value` function defined in the `FromMdbvalue` trait. The primary functionality of this code snippet is to convert a reference of `MdbValue` into another type `$t`. The type conversion is performed using `transmute` at line 4, which is included in `unsafe`. The function `new_from_size` (line 9) is used to create a new object `MdbValue` from the reference of generic type `T`. Therefore, the users of the package can create input for `from_mdb_value` with `new_from_size` function. All type conversion bugs occur in `from_mdb_value` function because of the problematic type conversion.

```
1  #[repr(align(2))]
2  #[derive(Copy, Clone, Debug)]
3  struct Padding { a: u8, b: u16, c: u8 }
4  fn main() {
5      let la = Padding { a: 10, b: 11, c: 12 };
6      let mdbval = MdbValue::new_from_sized(&la);
7      let res = i32::from_mdb_value(&mdbval);
8      println!("{:?}", res);
9  }
```

Listing 10: Exploit that trigger bug type II.

Type I: Misalignment bug. The first type of bug occurs when reinterpreting the type of the source object in memory to another type with a larger alignment. In the exploit code of Listing 9, we define and initialize an `i32` variable `a`, and convert it into `i64` using `from_mdb_value` defined in Listing 8, which would cause the misalignment issue. The data is aligned only if it is stored at an address that is a multiple of the type's alignment bytes. Most primitive types (e.g., u8 and u32 in this case) are aligned to their size. For example, a 32-bit integer (`i32`) should be stored at the memory address that is a multiple of 4. However, the starting address of `i32` may not be a multiple of 4, hence accessing a misaligned object can results in undefined behavior. In this case, a misaligned pointer dereference (Line 5 in Listing 8) would cause runtime panic. Generally, such undefined behaviors in architectures that do not support unaligned access (e.g., before ARMv5) would cause the program to crash.

Type II: Inconsistent layout bug. In Listing 10, the second type of bug occurs when reinterpreting the uninitialized area of the source object in memory. We define a struct `Padding` (line 3) and instantiate an object `la` (line 5), and then convert it into an `i32` primitive object `res`. It looks like the source and target objects share the same size (u8+u16+u8=i32), but there will be padding bits among each member variable in the struct for alignment. In this case, member `b`'s size is 16 bits (u16); thus, there will be 8-bit padding for both `a` and `c`. These paddings are uninitialized areas in memory, which would trigger the undefined behavior when `transmute()` accesses them. Besides, a further dangerous issue is the unknown padding layout in Rust. Different from the struct padding rule in C (i.e., `repr(c)`), which usually adds padding bits at the end of the struct, Rust has no guarantees of data layout made by the default representation (`repr(rust)`). That means the compiler can do whatever it wants to reorder fields based on access patterns. A possible rule in practice is to organize by field size to minimize padding. Therefore, the location of padding is random and may cause data exposure.

Type III: Mismatched scope bug. The third bug type happens when the value of the source object exceeds the bit-pattern range of the target type. The bit pattern refers to the raw binary representation of data in memory. In the case of Listing 11, we convert an `i32` variable into the `bool` type.

```
1  fn main() {
2      let a: i32 = 3;
3      let mdbval = MdbValue::new_from_sized(&a);
4      let res = bool::from_mdb_value(&mdbval);
5      println!("{:?}", res);  // illegal boolean type
6
7      let arr = [1u8; 2];
8      println!("{:?}", arr[res as usize]); // OOB index
9  }
```

Listing 11: Exploit that trigger bug type III.

However, the i32 has 2^{32} bit-patterns while the boolean type has only 2 bit-patterns (false/true). The value false has the bit pattern 0x00 and the value true has the bit pattern 0x01. Hence, an undefined behavior would occur if an bool object represents any other bit pattern. Moreover, even the conversion between same-sized types may suffer such an issue. For example, the string type in Rust only supports UTF-8 encoding that includes $(2^8 - 2)$ unit characters. When we convert an u8 (2^8 bit-patterns) into string type, the undefined behavior can also be triggered if the value of source object is 254 or 255. The third bug can also be exploited to trigger the Out-Of-Bound memory access (OOB). Originally, the compiler always inserts the bound check to protect us from the OOB vulnerability. When the compiler assumes that type has legal value and removes the unnecessary bound check, OOB can be triggered (see line 8).

B Root Cause of Type Confusion Bugs

Based on Table 3, we summarize the root cause of type confusion bugs we have found in two phases: First, we discuss how developers make mistakes based on type conversion patterns. Second, we study the error-prone methods of ❶ Conversion and ❷ Access, specifically on usages of unsafe functions.

Con → Con. TypePulse identifies more type confusion bugs in the concrete type conversion from the top 3,000 packages. In concrete type conversion, we highlight the causes of misalignment bugs since its number (21) is much more than the others (10 on inconsistent layout bugs and 6 on mismatched scope bugs). We consider the root cause to be the lack of alignment awareness. We can also find that developers suppress the warnings of alignment from Clippy [9]. While some developers consider the impacts of misalignment are minor since most operating systems nowadays can tolerate the unaligned memory access, we have discovered an issue that can cause to crash (see §5.5).

Gen → Con. In the type conversion *Gen → Con*, we have discovered more inconsistent layout bugs (15) than the others (2 on misalignment bugs and 4 on mismatched scope bugs). Based on our observation, the developers usually consider the input types that initialize the generic type have a stable

memory layout and consequently initialized. For example, the function as_byte_slice is always used to convert the generic type into the slice of u8, leading to uninitialized memory exposure.

Con → Gen. For bugs related to type conversion *Con → Gen*, we find that developers have tried to limit the input types by adding the size check, ensuring the memory layout to be stable. However, the size check is not sufficient to check the alignment and the validity of types. Nevertheless, we still consider that the developers of the top 3,000 packages provide more protection in this type conversion, leading to the least number (14) of bugs compared to the others (37 in Con → Con and 21 in Gen → Con).

❶ **Conversion.** For the methods used for type conversion, we find that developers make more mistakes with as than transmute. We assume that developers tend to use as more commonly since transmute is a unsafe function itself while as is not, but they are not aware that as can also create problematic types, even if it is a safe function. Since we find fewer numbers of mismatched scope bugs than the other two types of bugs, we consider that the maintainers of the top 3,000 packages are more experienced in avoiding this kind of bug. To support our conjecture, we randomly pick 10 more packages that are not ranked in the top 3,000 and find 6 more mismatched scope bugs. The bug discovered in lmdb-rs package is one of the examples (see Appendix A).

❷ **Access.** We also study the unsafe usages of problematic types which could trigger the bugs, and separate them into three categories. First, type conversion can cause bugs when developers try to build a slice or vector with unsafe functions such as from_raw_parts. Second, *dest* is a raw pointer type, and the developers try to dereference the raw pointer. The purpose of raw pointer dereference can be separated into two kinds: a) overwrite the value stored at the memory address and b) dereference the raw pointer to rebuild the reference. Third, developers try to use transmute between references, which is dangerous and might break the safety guarantee of reference.