Rusted Types: Static Detection of Rust Type Confusion Bugs

Zeyang Zhuang
The Chinese University of Hong Kong
Hong Kong SAR, China
zyzhuang22@cse.cuhk.edu.hk

Wei Meng
The Chinese University of Hong Kong
Hong Kong SAR, China
wei@cse.cuhk.edu.hk

Michael R. Lyu
The Chinese University of Hong Kong
Hong Kong SAR, China
lyu@cse.cuhk.edu.hk

Abstract

Rust is a modern programming language designed for strict memory and concurrency safety. Despite the safety checks, unsafe code in Rust can introduce memory safety issues. Unsafe type conversion, which reinterprets data from one type to another type, can alter the type spatial and temporal properties, leading to memory safety and data corruption issues. Since an unsafely converted value could violate a programmer's intended conversion semantics and allows attackers to access the sensitive memory, type confusion bugs have critical security implications. Despite the increasing number of Rust type confusion bugs, the problem has not been systematically studied and addressed by the Rust community.

In this paper, we propose DeRust, the first and a novel static analysis tool for detecting Rust type confusion bugs. By precisely tracking the inter-procedural data flows from unsafe type conversions to sensitive operations, DeRust can effectively and efficiently identify type confusion bugs. Our evaluation shows that DeRust can detect all the type confusion bugs in the benchmark datasets, and further helped report 54 previously unknown bugs in the top 1,000 Rust packages with 13 RustSec IDs assigned.

CCS Concepts

• **Security and privacy → Software security engineering**.

Keywords

Software Security; Program Analysis; Rust Security

ACM Reference Format:
Zeyang Zhuang, Wei Meng, and Michael R. Lyu. 2026. Rusted Types: Static Detection of Rust Type Confusion Bugs. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3744916.3764571

1 Introduction

Rust is a modern programming language designed for developing reliable and efficient software. It strictly enforces memory safety and thread safety through compile-time checks. Since memory safety issues in languages such as C and C++ have led to severe consequences and are difficult to be prevented, Rust becomes a good system programming language alternative. It has been integrated into prestigious open-source projects, such as the Linux kernel [34] and the Firefox browser [33]. The United States government has also advocated for the adoption of memory safety languages such as Rust to secure the nation's cyberspace [30].

Rust performs compile-time checks including the ownership and borrow checks, *etc.* to ensure memory and concurrency safety in the safe code [27]. Nevertheless, to provide more programming flexibility, Rust allows developers to write unsafe code for direct memory manipulation and performance-critical tasks. Existing studies [53, 65] have shown that unsafe code is the most common cause of memory safety issues in Rust programs.

Type confusion bugs are a type of memory safety issues in languages like C++ [51, 59]. The program could perform unsafe type conversion that downcasts a parent class object to one of its derived classes. Consequently, the memory can be corrupted as unknown data gets accessed after type conversion. We observe that unsafe type conversion also exists in Rust, resulting in Rust type confusion bugs and various unexpected program behaviors. Unsafe type conversion in Rust includes raw pointer casting, the `transmute` operation, and unsafe type conversion functions that reinterpret data from one type to another type.

Unsafe type conversion in Rust can alter the spatial and temporal type properties, leading to memory safety and logic issues. First, the spatial type properties (*i.e.*, size and value) can be manipulated and result in addition, deletion, or modification of sensitive data in memory. For example, converting a const `u8` pointer to a const `u32` pointer would add three bytes of unknown data at adjacent addresses and generate an uncertain value. Performing further operations on this value can trigger undefined behaviors such as out-of-bound memory access, *etc.* Similarly, altering the spatial type properties by deleting or modifying data can corrupt the data, leading to logic errors. Besides, unsafe type conversion can manipulate the type temporal property (*i.e.*, lifetime) by inappropriately extending the lifetime, resulting in dangling pointers.

In this work, we aim to systematically study and detect Rust type confusion bugs, which has not been done before to our best knowledge. We determine that a Rust type confusion bug is manifested in two steps: an unsafe type conversion, and data flows from the converted value to sensitive operations (*e.g.*, memory accesses) that cause undefined behaviors and even security breaches. Type confusion bugs could be exploited with nearly 100% reliability [1], primarily through illegal memory accesses. Based on the security consequences, Rust type confusion bugs can be classified into three categories: out-of-bound memory access, data corruption, and dangling pointer.

Rust type confusion bugs are severe and can lead to various security consequences such as memory corruption, crashes [40], use-after-free and double-free vulnerabilities, sensitive data leakage [58], and logic errors, *etc.* Experienced attackers can exploit them to achieve unauthorized access to sensitive data, control-flow

ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3764571

hijacking and even arbitrary code execution [11], and privilege escalation [31], *etc.* For example, we are able to successfully exploit some of the bugs we found. Our initial investigation reveals that from 2019 to 2023, RustSec advisories [36] reported 29 type confusion bugs. These all demonstrate that Rust type confusion bugs do exist in real-world programs and can pose significant risks, and motivate us to develop new techniques to effectively and efficiently detect such bugs.

There are two primary challenges in detecting type confusion bugs in Rust programs. First, there may exist complex and consecutive unsafe type conversions, along with intricate data flows of the converted values. Second, analyzing the diverse types of Rust and their type properties is rather challenging. For example, the generic type in Rust can accept any concrete type that adheres to its trait bounds; analyzing the type properties of generics and establishing comprehensive type analysis is not straightforward. Existing works on Rust bug detection [48, 49, 60] cannot address these challenges, as they are not specifically designed to detect type confusion bugs and have limited ability to reason complex Rust type features. Current dynamic analysis methods [20, 61] are also inappropriate due to the need to handle complex control, type and data constraints with real inputs, as well as the low efficiency.

We propose DeRust, a novel and precise Rust static analysis tool to detect type confusion bugs in Rust programs. It accurately captures the data flows of unsafely converted values, and tracks their subsequent uses in operations that cause undefined behaviors. To achieve this, we develop a novel *lattice-based* field- and context-sensitive inter-procedural alias analysis as well as a *type system analysis* for Rust, which enables the construction of precise static data-flow analyses for complex programs. We build a novel language model and abstract interpretation to accomplish precision and scalability. Besides, DeRust employs extensive language feature modeling that enables the analysis of the intricate type system in Rust, including generics and traits, algebraic data types, and standard library types, *etc.* Specifically, DeRust infers all concrete types of generic type according to the trait bounds to further analyze type properties. Based on the static data-flow and type system analysis, DeRust deploys various oracles to identify the three categories of type confusion bugs.

We implemented a prototype of DeRust with approximately 12,500 lines of code in Rust. To evaluate its effectiveness and efficiency in Rust type confusion bug detection, we synthesized 60 type confusion bugs and obtained a real-world dataset containing 29 bugs as our benchmark datasets. DeRust successfully detected all type confusion bugs in the benchmark datasets in a short amount of time while consuming moderate amount of memory. We further compared DeRust with three static analysis tools (Rudra [48], MIRChecker [60], and SafeDrop [49]) for detecting dangling pointer bugs, as well as with Clippy [8] for detecting type confusion bugs. DeRust performed the best in the comparison—other tools can hardly find the bugs in the datasets. The experiment results demonstrate DeRust's superb precision and efficiency in Rust type confusion bug detection. We then applied DeRust to detect unknown bugs in the well-maintained top 1,000 Rust packages ranked by download number on `crates.io`, and discovered 54 new bugs that we had manually confirmed. Specifically, it detected 27 out-of-bound memory access bugs, 22 data corruption bugs, and five dangling pointer bugs. These results further show the efficacy of DeRust's precise static data-flow analysis and accurate bug detection oracles. We further demonstrate the severe security consequences of those bugs through our manual exploitation. We responsibly disclosed all our findings to the relevant developers. At the time of writing, 42 bugs have been confirmed by the developers, six have been timely fixed, and 13 RustSec IDs have been assigned. We have received positive feedback from the Rust developer community. DeRust and the artifacts will be publicly available at https://github.com/cuhk-seclab/DeRust.

In summary, we make the following contributions:

- We are the first to systematically study and detect Rust type confusion bugs. We further demonstrate how such bugs can be exploited and their severe security consequences.
- We design and develop DeRust, the first and a novel precise static data-flow analysis tool to detect Rust type confusion bugs.
- With DeRust, we have detected 54 previously unknown bugs, among which 42 bugs have been confirmed by the developers with 13 RustSec IDs assigned.

2 Background

In this section, we present the relevant fundamental knowledge.

2.1 Rust Basics

Undefined behaviors in Rust. Undefined behaviors in Rust refer to operations or code patterns that, if executed, lead to unpredictable and potentially vulnerable results. Rust clearly lists out the scenarios that might trigger undefined behaviors [12], such as dereferencing dangling or misaligned pointers, index out-of-bound, modifying immutable bytes, and data races, *etc.* Undefined behaviors in Rust can lead to severe consequences such as memory corruption, or crashes. These outcomes can be further exploited by attackers to perform arbitrary code execution, or privilege escalation, thereby jeopardizing the integrity and safety of the program.

Rust type system. The Rust type system [43] is designed to ensure memory safety and concurrency safety. As Rust is a strongly typed language [35], each type possesses a rigorously defined data representation. Apart from the Rust built-in basic types, generic type and lifetime are also key features in the Rust type system. Generic type represents the abstract stands-in for concrete types or other properties within the trait bounds. Lifetime ensures that references to data are valid for a specified scope. The size and lifetime of a type belong to two distinct dimensions, namely the spatial and temporal properties. The size of a type determines the amount of memory occupied, and its lifetime indicates the validity period of the corresponding memory.

2.2 Type Conversion in Rust

Type conversion in Rust [46] refers to the process of converting a value from one type to another type. There are mainly three methods of type conversion in Rust, namely explicit conversion between types through the casting or `transmute` operation, custom conversion implementing special traits (*e.g.*, `From` and `Into` traits, *etc.*), and implicit coercion into a new type. Implicit coercion would be performed by the compiler without explicit code. Conversion

traits are typically handled by casting and `transmute` operations. Therefore, we introduce mainly casting and `transmute` operations.

Casting. Rust depends on the `as` keyword to perform explicit type casting. Casting operation in Rust can be either safe or unsafe. Safe type casting relies on rigorous static compiler checks, which involve bit manipulation or value adjustments. For example, it can lead to bit extension [14]. At line 5 in Listing 1, a `u8` type (1 byte) value `val` is casted to `u32` type (4 bytes). The Rust compiler safely ensure that `dst_ty1`'s most significant three bytes are properly set to 0. In contrast, casting between raw pointers can be unsafe. The raw pointer casting would reinterpret the bits of one type as another type without altering the bits of the original value. For example, in Listing 1, `src_ty` at line 3 is a const `u8` pointer. When it is converted to a const `u32` pointer using `as` casting, the Rust compiler would directly incorporate the value of three adjacent unknown bytes in memory. Consequently, `dst_ty2` is a misaligned pointer [29] pointing to an unknown value. This raw pointer casting is inherently unsafe, and subsequent operations on the `dst_ty2` pointer could lead to undefined behaviors.

Transmute. `Transmute` [41] is an unsafe operation in Rust. Essentially, `transmute` is equivalent to copying the bits of value from one type into another type. To use a `transmute` operation, the source and the destination types must have the same size, and the operation should be encapsulated in unsafe code. It allows for arbitrary type conversion by modifying the representations of bits without considering type compatibility. For example, at line 9 in Listing 1, `transmute` converts a const `u8` pointer into a const `u32` pointer. The validity of the original bits in the new type is not guaranteed, making `transmute` entirely unsafe and susceptible to causing undefined behaviors.

```
1  fn main() {
2      let val: u8 = 6;
3      let src_ty: *const u8 = &val as *const u8;
4      // Casting operation: as
5      let dst_ty1: u32 = val as u32;
6      let dst_ty2: *const u32 = src_ty as *const u32;
7      // Transmute operation: transmute
8      let dst_ty3 = unsafe { std::mem::transmute::<*const u8, *
         const u32>(src_ty) };
9      // Security consequences
10     let alias= unsafe { *dst_ty2 };
11     println!("{}", alias); // Panic! Misaligned pointer
         dereference: address must be a multiple of 0x4 but the
         current value is...
12     unsafe {
13         println!("{:p}", dst_ty2); // Disclose memory address
14         println!("{}", *dst_ty2); // Disclose unknown bytes
15         // Attackers can overwrite the unknown bytes with
       exploitation payload
16         let payload: u32 = 0xdeadbeef; // Payload can be
       shellcode, injected code, etc. to hijack control flow
17         std::ptr::write(dst_ty2 as *mut u32, payload); //
       Overwrite arbitrary sensitive memory with the payload
18         // Execute and successfully exploit...
19     }
20 }
```

Listing 1: An example of type confusion bug. It includes the type conversion by casting and `transmute` operations, and its potential security consequences.

3 Problem Statement

In this section, we describe type confusion bugs in Rust with real-world examples. Then we outline the research scope of our work, and discuss the research challenges.

3.1 Type Confusion Bugs

Type confusion bugs have been studied in the C++ programming language [51, 59] and domains such as OS kernel [55] and web browser [3]. Nevertheless, to the best of our knowledge, this is the first work that systematically studies type confusion bugs in the Rust programming language.

3.1.1 Problem Description. In this work, we define type confusion bugs in Rust as bugs that occur when there is an unsafe type conversion, and the value of the resulting type is used to perform operations that lead to undefined behaviors or potential security breaches. Accordingly, type confusion bugs manifest in two steps: ① *Unsafe type conversion*; ② *Operations causing undefined behaviors*.

Unsafe type conversion. Unsafe type conversion refers to reinterpreting data of one type as another type by unsafe operations. In this work, we specifically consider the following unsafe operations: raw pointer casting, `transmute`, and type conversion using unsafe functions (*e.g.*, `from_raw_parts` [19]). Unsafe conversions between different types would result in diverse type property manipulation behaviors, which we classify into four categories in Table 1. Our classification is based on whether the conversion would alter the size, value or lifetime of the data. Unsafe type conversion can manipulate the type's spatial properties (*i.e.*, size and value), leading to memory safety or logic issues. It can also change the temporal property (*i.e.*, lifetime), resulting in dangling pointers.

Table 1: The unsafe type conversion and operations causing undefined behaviors. These two steps constitute type confusion bugs.

Steps	Behaviors
Unsafe type conversion	Spatial manipulation: data addition
	Spatial manipulation: data deletion
	Spatial manipulation: data modification
	Temporal manipulation: lifetime extension
Operations causing undefined behaviors	Accessing a dangling place
	Accessing a misaligned place
	Producing an invalid value
	Indexing out-of-bound

Spatial manipulation: data addition refers to converting from a small-size type to a large-size type, which would incorporate additional *unknown* data in the memory. For example, line 6 in Listing 1 converts a 1-byte `u8` value to a 4-byte `u32` value by adding 3-byte unknown data at adjacent addresses. Performing further operations on this uncertain 4-byte value can trigger undefined behaviors, such as out-of-bound memory access, *etc.*

Spatial manipulation: data deletion can be categorized into two scenarios. First, converting from a large-size type to a small one can lead to data truncation or deletion, potentially causing logic

```
1  fn main() {
2      let src_ty: u8 = 6; // 7
3      let dst_ty = unsafe { std::mem::transmute::<&u8, &bool>(&
         src_ty) };
4      // let dst_ty = &src_ty as *const u8 as *const bool;
5      if *dst_ty {
6          println!("Go into the if block");
7      } else {
8          println!("Go into the else block"); // -> Oops!
9      }
10 }
```

Listing 2: An example leading to an unexpected result that unsafely converts types between same sizes from `u8` to `bool`.

```
1  #[repr(align(2))]   ......   #[repr(C)]
2  struct S {                   struct R {
3      a: u8,                       a: u8,
4      b: u16                       b: u16,
5  }                            }
6  fn main() {
7      let s = S { a: 6, b: 7 };
8      let r: R = unsafe { std::mem::transmute::<S, R>(s) };
9      let a: u32 = unsafe { std::mem::transmute::<S, u32>(s) };
10     println!("{:?}", s); // S { a: 6, b: 7 }
11     println!("{:?}", r); // R { a: 7, b: 5382 }
12     println!("{:?}", a); // 393223
13 }
```

Listing 3: An example that unsafely converts types between same sizes but different representations.

issues, such as erroneous data output. Second, converting between same-size types may also result in data loss when only parts of the data would be interpreted. For example, in Listing 2, a `u8` type is converted into a `bool` type at line 3 or 4. Although `u8` and `bool` have the same size (both occupying 8 bits), they convey different values through distinct type representations. If `src_ty` is 7, the program would take the if branch as expected. However, if `src_ty` is 6, the bool value `*dst_ty` would be evaluated as `False` and leads the program to incorrectly take the else branch. It is probably that the condition evaluation checks only the least significant bit (0 or 1), resulting in this counter-intuitive unexpected result, which we consider as control-flow divergence and a logic error.

Spatial manipulation: data modification involves both *data addition and deletion*. It may happen when some original data is discarded and some unknown data is added after converting into a type in the same size. In particular, some memory locations of a Rust `struct` may contain random uninitialized padding data for memory alignment. Different Rust `struct`s can have the same sizes and same fields but varied representations (*e.g.*, `repr(align)`, *etc.*) [32]. In other words, the underlying memory layouts of these `struct`s could differ. Accordingly, the conversion between these types would cause both data addition and data deletion. In Listing 3, there is an 8-bit padding for field `a` of `struct S`. If we convert `S` into a same-size type with a different memory layout, such as `struct R` or `u32`, the final value would include some random padding data and lose some original bits in `S`. As shown in the outputs at lines 11 and 12, the resulting variables `r` and `a` produce invalid random data. Such cases would further lead to other unexpected behaviors.

Temporal manipulation: lifetime extension happens when the lifetime of the resulting data is incorrectly extended after type conversion. In most cases, a raw pointer is converted into another type with an incorrect lifetime that becomes a dangling pointer when the original raw pointer is dropped. For instance, unsafe functions such as `from_raw_parts` would extend the lifetime of the resulting value by converting a raw pointer into another type; unsafe operations such as `&*` would convert a raw pointer into a reference with an unbounded lifetime [45]. Note that this problem is related to Rust memory safety issues such as use-after-free that have been extensively studied [48, 49, 60]. We study exclusively the lifetime extension problem caused by unsafe type conversion and do not address the other lifetime problems caused by incorrect drop trait implementation or erroneous lifetime annotation, *etc.*

Operations causing undefined behaviors. When a value created through an unsafe type conversion is used in certain operations, undefined behaviors would happen, resulting in a type confusion bug. Note that an unsafe conversion would not necessarily trigger any error and be regarded as a bug, if the resulting value is not further used or is used in only safe operations (*e.g.*, correct memory management to reclaim memory, *etc.*). Therefore, type confusion bugs only occur when further operations causing undefined behaviors are performed. Since Rust undefined behaviors cannot be exhaustively listed [12], in this work we cover the most common undefined behaviors as shown in Table 1.

3.1.2 Bug Definition. We define three categories of security-relevant Rust type confusion bugs based on the resulting buggy program behaviors, which pose security risks. There could be other categories of Rust type confusion bugs in theory. We discuss them in §7.

Out-of-bound memory access. As discussed in §3.1.1, the *data addition* spatial manipulation would create a value by accessing additional unknown data beyond the memory boundary of the original type, which falls under the classic out-of-bound memory access issue. Reading this value can potentially leak sensitive data in memory; by writing a new value to the converted larger-size data, the memory holding other important data can be corrupted. Besides, if this value with the upscaled type size is used for indexing operations, it can lead to index out-of-bound issues.

Data corruption. Data corruption may result from spatial type manipulation, in particular *data deletion and modification*. We exclude *data addition* as it would lead to out-of-bound memory access, which is more severe and is treated separately above. We consider two particular data corruption cases as discussed in §3.1.1, which would lead to logic errors. The first is that data modification might occur in conversion between same-size algebraic data types with different memory layouts, as illustrated in Listing 3. A data corruption bug happens when the converted value is accessed (and particularly read), as corrupted random data is used. The second is data deletion. It includes large-to-small unsafe type conversion. Data deletion can result from situations that primitive types such as `int`, `float` and `char` are converted to `bool` and only the least significant bit of the converted value is used in certain operations.

Dangling pointer. A dangling pointer is created when an unsafe type conversion incorrectly extends the lifetime of the resulting

value as discussed in §3.1.1. A dangling pointer can subsequently lead to use-after-free or double-free bugs.

3.1.3 Real-world bugs. We investigated the bug reports related to type confusion bugs in the RustSec advisory database [36] as of December 31, 2023. Type confusion bugs were reported annually, and there had been 29 such bugs in total. The result shows that type confusion bugs do exist in real-world projects and demonstrates the need of systematically studying and detecting type confusion bugs. We provide detailed statistics of these bugs including the advisory IDs and categorization in our artifact.

3.2 Research Scope

In this work, we aim to detect type confusion bugs in Rust programs and study the corresponding implications. Specifically, we target the three categories of security-relevant type confusion bugs we define. Other type confusion bugs are discussed in §7. Due to the type system disparities across different languages, the potential type confusion bugs through Rust Foreign Function Interface (FFI) [18] fall outside our research scope, and are left as future work.

Type confusion bugs are severe and could be exploited with nearly 100% reliability [1]. We use the example in Listing 1 to show the security implications. First, attackers can achieve unauthorized access to the memory (lines 13-14), which leads to sensitive information leakage [58]. The adjacent unknown memory can be stack data, heap data, or even users' confidential, *etc.* It can be leveraged to bypass protections like ASLR [10]. The access can trigger a program panic (lines 10-11) [40] when it violates the compiler safety checks, or causes a segmentation fault [37]. Second, attackers can overwrite the sensitive memory with exploitation payload (lines 15-18) [2, 3]. If the memory contains a function pointer, return address, or other executable data, it can be overwritten to redirect execution to existing code in the program, the shellcode, or malicious code, *etc.* In this work, we consider the automatic generation of exploitation for type confusion bugs in Rust programs as future work.

3.3 Research Challenges

To detect Rust type confusion bugs, there exist several challenges.

Complex unsafe type conversion. In Rust programs, there may exist complex and consecutive unsafe type conversions. Accurately capturing the data flow of the values through unsafe conversion and tracking their subsequent operations causing undefined behaviors are very challenging. The data flow should be tracked both intra- and inter-procedurally to confirm the spatial and temporal issues. It is non-trivial to achieve the analysis by interpreting the complex semantics of Rust instructions [28]. Establishing a good balance between efficiency and precision in bug detection is also difficult.

Rust type system analysis. Analyzing the diverse types of Rust requires tailored strategies. Particularly, the generic types are concretized either during the code generation phase through static dispatch or at runtime via dynamic dispatch [39]. Inferring their appropriate types and performing checks, whether during the compilation phase or after the code generation phase, is rather challenging. It is also necessary to conduct precise field-sensitive analysis of algebraic data types in Rust. One also needs to model and analyze common Rust standard library types [44] to establish a more comprehensive type analysis.

Dynamic analyses can hardly tackle this research problem, as complex control-flow, type and data constraints need to be satisfied with real inputs. Existing dynamic analysis approaches such as fuzzing [20] can hardly generate test cases to cover all possible execution paths and reach code locations that contain unsafe type conversions. Symbolic execution also suffers from critical limitations such as path explosion. It would also be time-consuming and resource-intensive for these approaches. Therefore, we think *static analysis* is a better approach to detecting type confusion bugs for its efficiency and scalability. However, existing static analysis tools [8, 48, 49, 60] can hardly address these research challenges, either, as they are not specifically designed for detecting type confusion bugs and have limited ability to reason complex Rust type features. Thus, a new solution needs to be proposed and developed.

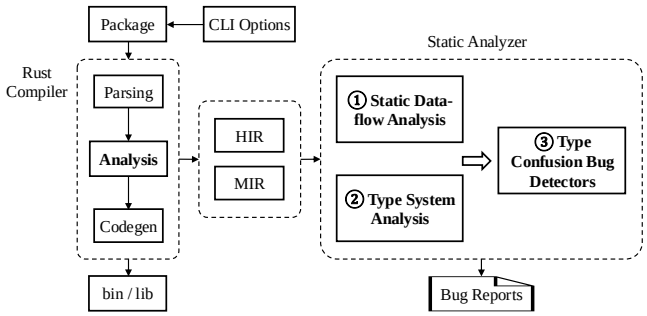

Figure 1: The architecture of DeRust.

4 Design

4.1 Overview

We propose DeRust, a precise static data-flow analysis tool for detecting type confusion bugs in Rust programs. We model type confusion bug detection as a data-flow problem [62], because such bugs are manifested when the data resulting from unsafe type conversion is used in operations causing undefined behaviors. To this end, DeRust performs the first *lattice-based field-sensitive and context-sensitive inter-procedural alias analysis* for Rust, as well as novel *type system analysis*. The architecture of DeRust is shown in Figure 1. The system consists of three parts. ① DeRust tracks the data flows on Rust compiler's internal IRs (§4.2). ② To allow DeRust to reason about the type properties, we model the type system of Rust (§4.3). ③ We design three bug detectors for detecting the three categories of type confusion bugs (§4.4).

4.2 Static Data-flow Analysis

We develop a static data-flow analysis to identify unsafe type conversion sites and track the data flow of the resulting converted value. This is not straightforward because a variable (which can be a struct field) holding the converted value may have multiple aliases across different function scopes. Therefore, it is crucial to construct the aliases in a flow-sensitive, field-sensitive and inter-procedural manner. We first define a novel language model in §4.2.1, and design

a new lattice-based alias analysis in §4.2.2 upon the language model. We further extend to a context-sensitive inter-procedural analysis and generate function summaries (§4.2.3) for better precision.

4.2.1 Language Model. We introduce our simplified language model that is built on top of the core syntaxes of Rust MIR [28] and HIR [22] rather than LLVM IR [26]. Its purpose is twofold. First, as a control-flow-graph (CFG)-based representation, MIR and HIR are well-suited for alias analysis and control-flow analysis to construct precise data flows. Second, they preserve type-related information, such as the Rust type system, which enables the identification of unsafe type conversion, the resulting values, and the operations causing undefined behaviors. We model the core syntax of the language model in Backus-Naur Form (BNF) [57] as shown in Table 2. We discuss the main syntax and its semantics as follows.

(1) The assignment statement ($p = r$) overwrites the value of left-hand expression p with a new value expressed by the right-hand expression r. The expression p is a **Place**, which can be a local variable v, a pointer dereference ($*v$), a qualified path representing a field access ($v.f$), or an array indexing ($v[i]$). The expression r is an **Rvalue**, which expresses operations performed on operands such as referencing ($\&p$), dereferencing ($*p$), computing the discriminant of the place, and type conversion, *etc.* For general Rvalue operations, the Rust type system requires that p and r must have the same type, hence a data flow from r to p can be established. However, when a type conversion is desired, the Rvalue operation *conv* is needed to support assignment operations between different types.
(2) The conversion operation (**CastKind**) converts values from one type (τ_1) to another type (τ_2). CastKind [14] supports both safe and unsafe type conversions. For example, `IntToFloat` and `FloatToInt` conversions are safely handled by the Rust compiler. We specifically retain the unsafe type conversion in CastKind, namely the `As` and `Transmute` operations. The `As` operation represents raw pointer casting, including `PtrToPtr` and `FnPtrToPtr`.
(3) We define several types of terminators. Function call ($Call(f, [op_1, \ldots], (p, b))$ invokes a function f with arguments $[op_1, \ldots]$. The return value is assigned to p in the basic block b. Goto ($Goto(b)$) unconditionally jumps to the successor basic block b. *Return* exits from the current function. SwitchInt ($SwitchInt(op, [b_1, b_2, \ldots])$) directs control flow to one of the basic blocks in the target list $[b_1, b_2, \ldots]$ conditionally on the value of the discriminant operand op. It matches the `if` and pattern matching constructs in Rust.

4.2.2 Alias Analysis. Based on the language model, we develop the first *lattice-based* flow- and field-sensitive alias analysis and extend it to a *context-sensitive* inter-procedural analysis in §4.2.3.

Abstract domain. We adopt a lattice scheme [56] to achieve efficient and scalable data-flow analysis. This differs from the traditional path-sensitive method in the literature [49]. We further discuss its limitations in §6.2. We employ the Tarjan algorithm [50] on the CFGs to separate the strongly connected components (SCCs) [62]. Each SCC contains a set of nodes representing basic blocks. Statements in one node are analyzed sequentially, which is flow-sensitive.

Our *abstract domain* specifically represents the alias sets of program variables. For each CFG, we denote the set of all variables as **Vars**. We define the abstract state **AState** as a lattice, which represents the alias set for each variable inside the function. Intuitively, the bottom element ($\bot$) is the initial state for a variable where no aliasing information is known. The top element ($\top$) represents the most imprecise state where any variable is aliased with any other variables. The partial order relation ($\sqsubseteq$) of the lattice is defined by the set inclusion operation. The join operation ($\sqcup$) is used to merge alias sets through the set union calculation. The abstract state at each program point is defined as a mapping from all the variables **Vars** to their respective **AState**s.

Table 2: Core syntax of DeRust's language model.

BasicBlock	b	$\in$	$\mathbb{Z}$
Type	τ	::=	`Bool`, `Char`, `Int`, `Uint`, `Float`, `Adt`, `Str`, `Array`, `Slice`, `Tuple`, `RawPtr`, `Ref`, `Param`, ... (Built-in types) | Generic type | Standard library types
Local	v	$\in$	$\{v_0, v_1, v_2, \ldots\}$
Function	f	$\in$	$\{f_0, f_1, f_2, \ldots\}$
Place	p	::=	$v \mid *v \mid v.f \mid v[i]$
Operand	op	::=	copy p | move p
CastKind	$conv$	::=	`As` | `Transmute`
Rvalue	r	::=	$op \mid \&p \mid *p \mid$ $conv(p, \tau_1, \tau_2) \mid disc(p, f)$
Statement	s	::=	$p = r$
Terminator	t	::=	$Call(f, [op_1, op_2, \ldots], (p, b)) \mid$ $Goto(b) \mid Return \mid$ $SwitchInt(op, [b_1, b_2, \ldots])$

Transfer functions. The *transfer functions* are defined as the set of instructions that can potentially change the alias states. We employ a unified alias state update strategy for each instruction. When we assign the Rvalue (*e.g.*, variable b) to the left Place (*e.g.*, variable a), they become aliases. We define several assignment statements in our language model, including general assignments (*e.g.*, `a=b`, `a=move b`), reference assignments (*e.g.*, `a=&b`), AddressOf assignments (*e.g.*, `a=&b as *const|*mut`), dereference assignments, type conversion assignments, *etc.* We also define terminators such as function call (*e.g.*, `a=Fn(b)`). Despite their different semantics, we uniformly apply the aliasing update rule ($S_a = S_a - a, S_b = S_b \cup a$), where S_i represents the set of aliases of variable i. Furthermore, due to the well-known challenges of pointer analysis [52], we approximate the alias relationships by disregarding the pointer arithmetic.

For algebraic data types, we apply a field-sensitive approach to enhance the precision. In Rust MIR, subfield identifiers are explicitly represented using indices, *e.g.*, `_0.0` indicates the first field of variable `_0`. We maintain the alias state for each subfield variable. During the execution of each instruction, when an alias set is updated, changes are synchronized to the alias sets of its subfields. Conversely, changes in a subfield do not update the parent alias state. For example, in Listing 4 the `foo` function generates new alias set `{_4, _0.0}`. The subfield `_0.0.0` is also updated to the alias set of subfield `_4.0`, resulting in `{_1, _3, _2, _4.0, _0.0.0}`.

Fixed-point algorithm. We implement a fixed-point algorithm to update the abstract states until a fixed point is reached. We adopt the worklist algorithm [62], where the worklist W is initialized to contain all the SCCs in the CFG. The state is first updated by joining the states of all the predecessors of a SCC. Next, we apply its transfer

```
struct S { A { ptr: *mut u32 }, B }
fn foo(_1: &mut u8) -> S {
bb0:
    _3 = &raw mut (*_1); // alias set: {_1, _3}
    _2 = move _3 as *mut u32 (PtrToPtr); // alias set: {_1,
     _3, _2}
    (_4 as A).0 = move _2; // alias set: {_1, _3, _2, _4.0}
    _0.0 = move _4; // alias sets: {_4, _0.0}, {_1, _3, _2,
     _4.0, _0.0.0}
    return;
}
fn main() {
bb0:
    _1 = const 6_u8; // alias set: {_1}
    _2 = &mut _1; // alias set: {_1, _2}
    _3 = foo(_2) -> [return: bb1, unwind continue]; // alias
     set: {_1, _2, _3.0.0}
bb1: ...
}
```

Listing 4: An example MIR code for alias analysis and inter-procedural analysis.

functions to update the state of the SCC. If the state changes, all the successors will be inserted into the worklist for re-analysis. We handle loops by identifying SCC, each of which captures the cyclic control flow of a loop and enables iterative fixed-point computation until convergence. We set the fixed-point threshold to 100 based on most static analyzers' practices. This ensures termination in complex loops while allowing simple loops to converge well before reaching the threshold. The algorithm terminates when either the state stabilizes in the lattice, or the worklist W becomes empty.

4.2.3 Inter-procedural Analysis. We extend the data-flow analysis to a *summary-based context-sensitive* inter-procedural one. A callee function is analyzed only if it resides within the current package being examined and its location can be statically determined. If it originates from dependency packages or is resolved at runtime (*i.e.*, dynamic dispatch through a vtable [4]), the function call would be skipped for simplicity. We consider data-flow analysis for these cases as future work. Since in real-world Rust repositories, most functions are implemented using static dispatch, our analysis is relatively comprehensive. Additionally, foreign functions called through FFI are not within our analysis scope.

To avoid duplicated analysis of one function, we design the *summary-based* context-sensitive method [62, 63]. It caches previously computed results (*i.e.*, function summary) in a lookup table cache: $((f, in\_state), out\_state)$ that maps a calling context (f, in_state) to an output out_state. f represents the function name, in_state represents the alias states of the function arguments, and out_state is the function summary.

The function summary contains two types of information. First, it captures the alias relationships among function arguments and the return value. This can be used to update the alias state of the caller function in a field-sensitive manner. Taking Listing 4 as an example, the function summary of function `foo` is `{_1, _0.0.0}`. When the `main` function calls the `foo` function, the alias set will be updated to `{_1, _2, _3.0.0}`. Second, the function summary tracks the occurrences of unsafely converted values within function arguments and return value. As demonstrated in Listing 4, the variable `_2` in function `foo` is unsafely converted. Since the return value `_0.0.0` is an alias of `_2`, it is also marked. Recording such information helps detect type confusion bugs in §4.4.

4.3 Type System Analysis

To detect type confusion bugs, we need to model the spatial and temporal properties of types, such as computing the type size, identifying the memory layout [32], and comparing types. Type system analysis in DeRust involves two parts. First, it infers all the concrete types of the generic types based on their trait bounds. Second, it encodes all built-in types and other essential types in Rust for comprehensive type comparison.

Generics and trait analysis. By examining the trait bounds, our analysis infers the set of appropriate concrete types to substitute the generics. In Rust, there are primarily two trait declaration methods [24]. The first uses pre-defined built-in traits, such as `Debug`, `Copy`, and `Clone`, *etc.* Many basic Rust types including primitive types, have implemented these traits. We collect a comprehensive list of built-in traits based on the language items of Rust [25]. Then we enumerate and collect all the types that implement these built-in traits, including the basic types and custom types in the Rust programs. The second declares custom traits, which in turn require developers to design custom types to implement. Regardless of whether the trait employs static dispatch or dynamic dispatch [39], types that implement this trait would be included as candidate types for the generics. Specifically, DeRust locates trait bounds through the HIR `GenericBound` syntax [21], and visits all the HIR items to identify the types that implement these traits. We extend this concrete type set collected in HIR to also MIR, therefore establishing the mapping of generics between MIR and HIR.

Type encoding. We collect information about types to facilitate type comparison. The developer-annotated attributes (*e.g.*, `repr`) are used to supplement the type layout information. DeRust encodes all the built-in types and custom types. For other types in the Rust standard library, such as `String`, `Vec`, `Box`, *etc.*, they cannot be directly represented using the internal `TyKind` syntax [42] of Rust MIR. To address this issue, we employ symbolic matching on top of the MIR to encode these types and all their associated API operations (*e.g.*, `Vec::new`, `String::from`, `Box::new`, *etc.*). We extend these type information to our type system.

4.4 Type Confusion Bug Detectors

We design three type confusion bug detectors according to the bug definition in §3.1.2. The workflow of the detectors involves two main parts. First, unsafe type conversions are identified. Second, a corresponding oracle is invoked for checking buggy behaviors. Specifically, the detectors consider four unsafe type conversion scenarios: `(Con` $\rightarrow$ `Con)`, `(Con` $\rightarrow$ `Gen)`, `(Gen` $\rightarrow$ `Con)`, and `(Gen` $\rightarrow$ `Gen)`, where `Con` represents concrete types and `Gen` refers to the generic type. By obtaining the available set of concrete types for a generic type, all types in the set are enumerated to replace generics for approximation. Consequently, an unsafe type conversion is transformed into the `Con` $\rightarrow$ `Con` scenarios, which are used for subsequent detection. The respective details are as follows.

Out-of-bound memory access detector. When an unsafe type conversion between different concrete types is detected, the detector compares the sizes of the two types to identify small-to-large type conversion. The detector labels the resulting value and its field-sensitive alias sets. The detector then reports an out-of-bound memory access bug by checking whether any of these values are used in specific operations, including memory accesses such as dereferencing values and accessing (loading from or storing to) places, element indexing (*e.g.*, array indexing), unsafe callee functions (*e.g.*, `ptr::read`, `ptr::write`, and `intrinsics::copy`, *etc.*).

Data corruption detector. This detector first detects unsafe spatial type manipulations that result in data deletion or modification. It identifies unsafe conversions from an algebraic data type with paddings to another type with different memory layouts. Besides, it identifies large-to-small unsafe type conversions, and primitive type conversions (*e.g.*, `int`, `float`, *etc.*) into `bool`. It reports a data corruption bug by checking whether the resulting values and aliases are used in operations or unsafe functions for memory access.

Dangling pointer detector. This detector inspects all unsafe type conversion functions (such as `from_raw_parts`, *etc.*) and syntaxes (such as `&*`) that extend the lifetime of the resulting value. It reports a dangling pointer bug by verifying whether the resulting value and its aliases are used in operations such as memory accesses (*e.g.*, dereferencing and accessing place) or unsafe callee functions (*e.g.*, memory access or `drop`), after the original value has been dropped.

5 Implementation

We implemented a prototype of DeRust with approximately 12,500 lines of Rust code on top of Rust compiler (`rustc` 1.76.0-nightly). Due to the fact that MIR is a control-flow graph-based representation [28], while LLVM IR is a Static Single Assignment (SSA) based representation [26], off-the-shelf LLVM toolchains and static analysis algorithms cannot be directly applied to our problem domain. Our analysis algorithms and tools such as static data-flow analysis, type system analysis, and the bug-reporting system, are *designed and implemented from scratch* for DeRust, which is non-trivial. Specifically, to construct an efficient data-flow analysis, DeRust terminates the algorithm when the number of fixed-point iterations exceeds a configurable threshold. To calculate the type sizes, we utilize the `SizeSkeleton` syntax [38] in Rust MIR and evaluate the actual memory object the type points to.

6 Evaluation

In this section, we first evaluate the capability of DeRust on a benchmark dataset we construct in §6.1, and compare DeRust with other tools in §6.2. We then assess the unknown bug detection ability and the security implications of these bugs in §6.3, and elaborate on the false positives in §6.4. We finally discuss the root causes and the mitigations of type confusion bugs in §6.5.

Experiment setup. We conduct experiments on a machine with a 3.60 GHz 8-core Intel Xeon W-2223 CPU and 32 GB of memory running Debian GNU/Linux 11.

Table 3: Comparison results with other existing tools.

Tools	Category 1	Category 2	Category 3	Total
DeRust	10	13	6	29
Clippy	4	2	-	6
Rudra	-	-	3	3
MIRChecker	-	-	6	6
SafeDrop	-	-	6	6

6.1 Benchmark Evaluation

Evaluating the bug detection capability of DeRust needs a test suite with ground truths. We thus took the first step to manually synthesize a benchmark dataset for the Rust type confusion bugs. Our synthetic benchmark consists of 60 test cases, covering all three categories of type confusion bugs. We created the benchmark for each category of the bugs in reference to our definitions in §3.1.2. DeRust is capable of detecting all type confusion bugs in the benchmark with no false positive or false negative. It can detect each of these bugs within one second. The details of the synthetic dataset are discussed in our artifact.

We further conducted experiments to detect currently known type confusion bugs on a real-world benchmark collected from the RustSec advisory database, which is introduced in §3.1.3. To reflect the essence of each bug, we manually downloaded all the corresponding packages and switched them to the versions where the bugs occurred. We also applied DeRust to these packages and found that all the existing type confusion bugs can be successfully detected. The experiments demonstrate that DeRust has no false positive or false negative in the ground-truth datasets.

We also measured the elapsed time and peak memory usage by DeRust for analyzing each package. To detect these 29 type confusion bugs, DeRust took an average of 5.48 seconds and a maximum of 42.96 seconds. The average and maximum memory consumption were 43.44 MB and 570 MB, respectively. As observed, the analysis for each package was completed in a short amount of time and consumed reasonable amount of memory. This demonstrates the effectiveness and scalability of DeRust.

6.2 Comparison with Existing Tools

To our best knowledge, we are the first one to develop a systematic bug detection tool to detect Rust type confusion bugs. Consequently, we are unable to find similar techniques to conduct apple-to-apple comparisons. Since type confusion bugs encompass the lifetime extension issues caused by unsafe type conversion, which leads to the dangling pointer bug, we select the techniques for detecting the lifetime issues for comparison, which are Rudra [48], MIRChecker [60], and SafeDrop [49]. We also incorporate the official static linter Clippy [8]. Our comparison criterion is to check whether these tools can detect the bugs of the real-world benchmark from the RustSec advisories. We reuse the same experiment setup in §6.1. Specifically, we followed the identical experiment configurations and running procedures as described in these artifacts. The comparison results are shown in Table 3. *DeRust outperformed all other tools in detecting type confusion bugs.* We discuss the details as follows.

Comparison with Clippy. Clippy [8] is a static analysis tool that implements over 700 types of lints to detect common mistakes

in Rust programs. We found that Clippy supports two relevant types of lints to detect unsafe type conversions. First, the `cast_ptr_alignment` lint [15] checks whether raw pointer casting can create a misaligned pointer. Second, the `unsound_collection_transmute` lint [47] detects `transmute` between collections. The Clippy version used is `rustc` 1.76.0, and it detected six bugs. We identify two of Clippy's limitations. First, Clippy lacks oracles for identifying type confusion bugs. It can only detect limited unsafe type conversion and cannot construct data flows. Second, Clippy does not support the analysis of generic types. Furthermore, it has been observed that developers frequently disregard the warnings generated by Clippy due to a significant number of false positives.

Comparison with Rudra. Rudra is a bug detector designed to identify memory safety bugs such as dangling pointers. To conduct a fair comparison, we migrated Rudra from the old `rustc` 1.56 [7] to the newer `rustc` 1.76.0. This is because over the past years, the Rust compiler has undergone several major updates, introducing many new features. The migration process is non-trivial. Our evaluation results show that Rudra was able to identify only three dangling pointer bugs. The main reason is that Rudra's static analysis framework is relatively simple, lacking static data-flow analysis such as alias analysis and inter-procedural analysis. Rudra is not as effective as DeRust in detecting dangling pointer bugs.

Comparison with MIRChecker. MIRChecker is a static analysis tool that detects memory safety bugs such as integer overflow and dangling pointers. Its implementation [5] is based on `rustc` 1.56. Due to the complexity of the implementation, migrating it to the latest Rust compiler requires significant manual effort and may even be infeasible [6]. Thus, we conducted our comparison on the old compiler version. MIRChecker detected all six dangling pointer bugs. Nevertheless, MIRChecker does not implement inter-procedural analysis. The authors also explained that the false-positive rate can reach around 80% to 90% [60].

Comparison with SafeDrop. SafeDrop is a static analysis tool to detect dangling pointer bugs. We performed a comparison using its latest implementation [9] on `rustc` 1.75.0. SafeDrop detected six dangling pointer bugs. However, due to its design of path-sensitive analysis, SafeDrop would traverse all possible execution paths of the program, which is rather expensive. In many cases, it would suddenly abort the analysis. Our experimental results revealed that it failed to analyze complex packages because of the path explosion issue. Besides, SafeDrop cannot analyze complex Rust types such as `Vec`, `Box`, *etc.* It suffers from a high false-positive rate for its low implementation quality. The authors claim that they are still fixing the bugs and it is under heavy development. This demonstrates that our lattice scheme is more effective and scalable compared to a path-sensitive approach.

6.3 Unknown Bug Detection

We further apply DeRust to detect unknown type confusion bugs in the real world for its superb bug detection performance. The other tools mentioned above can hardly detect type confusion bugs and are *excluded* from the experiment. *Our experiment shows that DeRust is able to find many unknown type confusion bugs in well-maintained Rust programs.* We collect real-world Rust packages from `crates.io`, the Rust official crate registry. We filter out packages that are related to multi-threading, asynchronous programming, and FFI, since these are not our concern. We download the top 1,000 packages ranked by download number. The code quality of these packages is relatively high, and they are frequently updated or maintained. Regarding package size [13], the largest package contains 18,707k LoC, while the average package size is 39k LoC.

We applied DeRust to these 1,000 packages, and it detected a total of 54 true-positive type confusion bugs. It took four hours to finish the experiment, demonstrating the significant advantages of DeRust over dynamic analysis methods.

We categorize these type confusion bugs and show the results in Table 4. The complete list of all detected bugs is presented along with our artifact. DeRust found type confusion bugs from heavily tested packages, some of which contain unit tests with extensive code coverage. The detected bugs are non-trivial as they have existed for several years. This indicates that even expert Rust developers may write error-prone code that triggers type confusion bugs. We discuss the developer feedback, some bug cases, and the exploitability as follows.

Table 4: The results of bug detection on top 1,000 packages.

Bug	Detected	Confirmed
Category 1	27	25
Category 2	22	15
Category 3	5	2
Total	54	42

```
1  // spl/token-swap/program/src/instruction.rs#L605-L612
2  pub fn unpack<T>(input: &[u8]) -> Result<&T, ProgramError> {
3      ...
4      let val: &T = unsafe { &*(&input[1] as *const u8 as *
         const T) }; ...
5  }
6  // Proof of Concept for exploitation
7  use spl_token_swap::instruction::unpack;
8  fn main() {
9      let val: [u8; 5] = [42; 5];
10     let dst = unpack::<u32>(&val).unwrap();
11     let dst_ptr = dst as *const u32;
12     unsafe { println!("{:p}", dst_ptr); } // Disclose memory
13     println!("{}", dst); // Trigger program panic
14 }
```

Listing 5: The type confusion bug detected in spl-token-swap package and the proof of concept of exploitation.

Bug disclosure and developer feedback. We reported all the bugs to the relevant developers. At the time of writing, 42 bugs have been confirmed by the developers, six have been fixed, and 13 RustSec IDs have been assigned. The main reason why some bugs have not been confirmed is that we are still awaiting further responses from the developers. We are also in the process of reporting the confirmed bugs to RustSec advisories and the CVE program. Developers of Rust packages such as *cxx* and *anyhow* have communicated the importance of DeRust. We believe that by expanding the experiment scale, DeRust will likely find more bugs.

Bug cases. We describe some type confusion bug cases. First, DeRust detected 27 out-of-bound memory access bugs and 22 data corruption bugs. Out-of-bound memory access bugs result from

```
1 // bytemuck/src/internal.rs#L321
2 pub(crate) unsafe fn try_cast_mut<A: Copy, B: Copy> ( ...
3     if align_of::<B>() > align_of::<A>()
4       && !is_aligned_to(a as *const A as *const (), align_of
        ::<B>()) { ...
5     } else if size_of::<B>() == size_of::<A>() { ...
```

Listing 6: The false positive case in bytemuck package.

```
1 // semver/src/identifier.rs
2 pub(crate) struct Identifier { head: NonNull<u8>, ...
3 // SAFETY: repr must be in the inline representation, ...
4 unsafe fn inline_as_str(repr: &Identifier) -> &str {
5     let ptr = repr as *const Identifier as *const u8; ...
```

Listing 7: The false positive case in semver package.

cases where developers convert `u8` type to larger integer types such as `u32`, `u64`, or custom types such as `BStr` or `__m128i`, *etc.* Data corruption bugs occur in several cases. These involve developers using unsafe conversions for data truncation, such as converting a larger type such as `Vec<u32>`, `Chunk` into `u8`, and performing type conversion that changes the layout of structures such as `Object`.

In Listing 5, a Rust code example is presented. An unsafe type conversion occurs at line 4, where `u8` type is converted to `T`, which can be replaced by many other types. Specifically, if the size of `T` is larger than `u8`, it may trigger an out-of-bound memory access bug. If `T` is `bool`, it can trigger a data corruption bug. The developer acknowledged this as an unsound implementation and responded that they would accept a pull request with proper implementation. We are currently working on the fix with the developers.

DeRust detected five dangling pointer bugs. The primary scenario is the unsound usage of `Vec::from_raw_parts` function. Besides, an allocation failure would result in a dangling pointer, which could create new slice with `slice::from_raw_parts`.

Exploitability and security implications. As presented in §3.2, Rust type confusion bugs can be reliably exploited by attackers and lead to severe security consequences. We present a proof of concept towards one bug we detected in spl-token-swap package in Listing 5. By leveraging the out-of-bound memory access bug, we can disclose the sensitive data, and bypass security mechanisms such as ASLR [10] or DEP [16], *etc.* We further achieve a program panic [40]. In some complex scenarios, program crashes and panics can lead to denial-of-service (DoS) [17]. The sensitive data can also be overwritten with payload to accomplish arbitrary code execution, or privilege escalation, *etc.* In theory, all these type confusion bugs can be exploited by experienced attackers.

6.4 False Positive Analysis

After carefully examining the diagnostic messages, the false positive rate of DeRust reaches around 36%. We find there are several reasons that cause the false positive cases. First, the nature of static analysis inevitably introduces imprecision, since the algorithm of DeRust is to provide an over-approximation of program execution. Besides, we describe two more reasons related to code patterns that we summarize based on the source codes.

Assertion checks. DeRust misses assertion checks added by the developers, leading to false positives. As shown in Listing 6, the bytemuck package performs generic type alignment checks at lines 3 and 4. Similarly, a type size mismatch check is performed at line 5. These checks prevent the out-of-bound memory access bug. DeRust cannot analyze the semantics of these assertion checks. The meanings of different assertion checks vary across various packages, making it challenging to generalize their patterns. Therefore, we manually verify and discard these false positives.

Context-sensitive semantics. DeRust also struggles to accurately analyze complex contextual semantics. As illustrated in Listing 7, an `Identifier` pointer is converted to a `u8` pointer at line 5. However, the actual memory pointed to by the `Identifier` pointer is the first field of the `struct`, which is type `u8`. This conversion is consequently safe. DeRust does not support complex contextual semantics analysis such as pointer analysis. Additionally, at line 3, there are comments related to safety declarations. However, DeRust is unable to analyze such semantic comments as well. We found that assertion checks account for around 90% of the false positives, while the remaining 10% are due to context-sensitive semantics. This suggests that some developers attempt to employ assertion checks to mitigate type confusion bugs.

6.5 Understanding Type Confusion Bugs

Root causes. We summarize the root causes of type confusion bugs. When performing unsafe type conversions, we found that developers make more mistakes using `as` for raw pointer casting compared to using `transmute`. This is because `transmute` is widely known as unsafe, and developers carefully use it. In contrast, raw pointer casting is a native operation in Rust that can be used in safe code. However, raw pointer casting is unsafe and can change the spatial and temporal properties of type. This lack of unsafety awareness such as being wrapped in unsafe code makes developers more likely to write error-prone codes when using raw pointer casting.

Mitigations. As a systematic detection tool for Rust type confusion bugs, DeRust can help prevent such errors. Developers should apply appropriate countermeasures when performing unsafe type conversions. For example, they can utilize assertion checks and error handling, as shown in Listing 6. When dealing with unsafe conversions involving generic types, setting appropriate and correct trait bounds is also important. We attest that we responsibly helped the corresponding parties (*e.g.*, the developers) initiate the mitigation process as early as possible.

7 Discussion

Other type confusion bugs. We study three categories of security-relevant Rust type confusion bugs. However, there may exist other cases. As illustrated in Listing 2, unsafe type conversion can lead to control flow divergence. A specific oracle should be added to detect this issue. We incorporated this oracle and conducted additional experiments to analyze the packages. The results did not reveal any control flow divergence issue. Another type of behavior namely integer overflow [23] requires complex numerical analysis and is considered future work. Type confusion bugs can also occur when interacting with the FFI. We have implemented DeRust in a highly

extensible manner. Researchers can easily and conveniently add new testing oracles on top of our tool in the future.

Limitations and future work. DeRust does not support pointer analysis. This would miss some execution paths and result in a simplified construction of aliases, which may lead to some false negatives. Developing a sound pointer analysis for Rust is rather challenging due to its rich type system. We consider this as future work. Besides, utilizing FFI to interact with other languages has become popular in Rust. Therefore, considering and analyzing the type conversions and their consequences through FFI boundaries is a good research direction. Due to the disparities in type systems, this also requires additional work, such as considering LLVM IR. We leave these as future work. Besides, we consider dynamic validation of DeRust's static analysis results and the automatic exploitation generation as future work.

8 Related work

Type confusion bugs. Type confusion bugs have been studied in other programming languages, such as C++ [51, 54, 59] and JavaScript [64]. Tools such as Caver [59], HexType [54] and TypeSan [51] have been proposed to detect type confusion bugs in C++. These tools are designed for LLVM IR, where many Rust language features are not preserved, making it difficult to precisely detect unsafe type conversions and therefore Rust type confusion bugs. Our research demonstrates that type confusion bugs can also occur in Rust. Type confusion bugs in Rust and those in C/C++ are distinct research problems. Their root causes, consequences, and bug patterns are significantly different.

Other Rust bugs. There are several works [53, 65] investigating memory safety bugs in Rust. Rudra [48] identifies three bug patterns related to memory and concurrency bugs to discover them. MIRChecker [60] utilizes static analysis and constraint solving to model numerical and symbolic values, thereby detecting integer overflow and dangling pointer bugs. SafeDrop [49] proposes path-sensitive data-flow analysis to detect dangling pointer bugs. In contrast, we detect Rust type confusion bugs that have not been studied, which can pose severe security risks.

9 Conclusion

We propose DeRust, the first and a novel precise static analysis tool designed for detecting Rust type confusion bugs that have not been systematically studied. By precisely tracking unsafe type conversion and analyzing further operations on the converted values, DeRust effectively detects type confusion bugs that could lead to memory safety or logic issues, and critical security breaches. DeRust helped find 54 bugs in popular real-world Rust programs and 42 bugs have been confirmed, with 13 RustSec IDs assigned.

Acknowledgments

The authors would like to thank the authors of MIRChecker, Dr. Zhuohua Li and Dr. Maoli Liu, for their valuable suggestions and feedback. The work described in this paper was partially supported by two grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No.: CUHK 14209323; and Project No. SRFS2425-4S03 of the Senior Research Fellow Scheme).

References

[1] 2015. Project zero at Google. https://googleprojectzero.blogspot.com/2015/06/what-is-good-memory-corruption.html.
[2] 2018. HackSysTeam Windows Kernel Vulnerable Driver: Type Confusion Vulnerability Exploitation. https://hackingportal.github.io/Type_Confusion/type_confusion.html.
[3] 2022. Chromium super (inline cache) type confusion. https://github.blog/2022-06-29-the-chromium-super-inline-cache-type-confusion/.
[4] 2023. Rust Deep Dive: Borked Vtables and Barking Cats. https://geo-ant.github.io/blog/2023/rust-dyn-trait-objects-fat-pointers/.
[5] 2024. Mirchecker artifact. https://github.com/lizhuohua/rust-mir-checker.
[6] 2024. Mirchecker artifact issue 12. https://github.com/lizhuohua/rust-mir-checker/issues/12.
[7] 2024. Rudra artifact. https://github.com/sslab-gatech/Rudra.
[8] 2024. Rust Clippy. https://github.com/rust-lang/rust-clippy.
[9] 2024. Safedrop artifact. https://github.com/Artisan-Lab/SafeDrop.
[10] 2025. Address space layout randomization. https://en.wikipedia.org/wiki/Address_space_layout_randomization.
[11] 2025. Arbitrary code execution in Wikipedia. https://en.wikipedia.org/wiki/Arbitrary_code_execution.
[12] 2025. Behavior considered undefined. https://doc.rust-lang.org/reference/behavior-considered-undefined.html.
[13] 2025. Cargo loc in crates. https://crates.io/crates/cargo-loc.
[14] 2025. CastKind in rustc_middle::mir. https://doc.rust-lang.org/stable/nightly-rustc/rustc_middle/mir/enum.CastKind.html.
[15] 2025. Cast_ptr_alignment in Rust Clippy. https://rust-lang.github.io/rust-clippy/stable/index.html#/cast_ptr_alignment.
[16] 2025. Data execution prevention. https://learn.microsoft.com/en-us/windows/win32/memory/data-execution-prevention.
[17] 2025. Denial-of-service attack. https://en.wikipedia.org/wiki/Denial-of-service_attack.
[18] 2025. Foreign Function Interface in the Rustonomicon. https://doc.rust-lang.org/nomicon/ffi.html.
[19] 2025. From_raw_parts in Rust standard library. https://doc.rust-lang.org/std/index.html?search=from_raw_parts.
[20] 2025. Fuzzing with Cargo-fuzz. https://github.com/rust-fuzz/cargo-fuzz.
[21] 2025. GenericBound in rustc_hir::hir. https://doc.rust-lang.org/stable/nightly-rustc/rustc_hir/hir/enum.GenericBound.html.
[22] 2025. The High-level Intermediate Representation (HIR). https://rustc-dev-guide.rust-lang.org/hir.html.
[23] 2025. Integer overflow example in Ariane flight V88. https://en.wikipedia.org/wiki/Ariane_flight_V88.
[24] 2025. Keyword Trait. https://doc.rust-lang.org/std/keyword.trait.html.
[25] 2025. Language items in Rust. https://github.com/rust-lang/rust/blob/master/compiler/rustc_hir/src/lang_items.rs.
[26] 2025. LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html.
[27] 2025. Meet Safe and Unsafe in the Rustonomicon. https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html#meet-safe-and-unsafe.
[28] 2025. The Mid-level Intermediate Representation (MIR). https://rustc-dev-guide.rust-lang.org/mir/index.html.
[29] 2025. Places based on misaligned pointers. https://doc.rust-lang.org/reference/behavior-considered-undefined.html#places-based-on-misaligned-pointers.
[30] 2025. Press Release: Future Software Should Be Memory Safe. https://www.whitehouse.gov/oncd/briefing-room/2024/02/26/press-release-technical-report/.
[31] 2025. Privilege escalation. https://en.wikipedia.org/wiki/Privilege_escalation.
[32] 2025. Representations in Type Layout. https://doc.rust-lang.org/reference/type-layout.html#representations.
[33] 2025. Rust for Firefox. https://wiki.mozilla.org/Oxidation.
[34] 2025. Rust for Linux. https://github.com/rust-for-linux.
[35] 2025. Rust is strongly typed and statically typed. https://en.wikipedia.org/wiki/Rust_(programming_language)#Types.
[36] 2025. RustSec Advisory Database. https://rustsec.org/advisories.
[37] 2025. Segmentation fault. https://en.wikipedia.org/wiki/Segmentation_fault.
[38] 2025. SizeSkeleton in rustc_middle::ty. https://doc.rust-lang.org/stable/nightly-rustc/rustc_middle/ty/layout/enum.SizeSkeleton.html.
[39] 2025. Static and Dynamic Dispatch of Rust Trait Objects. https://doc.rust-lang.org/1.8.0/book/trait-objects.html.
[40] 2025. To panic! or Not to panic! https://doc.rust-lang.org/book/ch09-03-to-panic-or-not-to-panic.html.
[41] 2025. Transmute in std::mem. https://doc.rust-lang.org/std/mem/fn.transmute.html.

[42] 2025. TyKind in rustc_middle::mir. https://doc.rust-lang.org/stable/nightly-rustc/rustc_middle/ty/sty/type.TyKind.html.

[43] 2025. Type system. https://doc.rust-lang.org/reference/type-system.html.

[44] 2025. Types in the Rust Standard Library. https://doc.rust-lang.org/std/index.html#modules.

[45] 2025. Unbounded Lifetimes in the Rustonomicon. https://doc.rust-lang.org/nomicon/unbounded-lifetimes.html.

[46] 2025. Understand type conversions. https://www.lurklurk.org/effective-rust/casts.html#item-5-understand-type-conversions.

[47] 2025. Unsound_collection_transmute in Rust Clippy. https://rust-lang.github.io/rust-clippy/stable/index.html#/unsound_collection_transmute.

[48] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. 2021. Rudra: Finding memory safety bugs in rust at the ecosystem scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 84–99.

[49] Mohan Cui, Chengjun Chen, Hui Xu, and Yangfan Zhou. 2023. SafeDrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis. *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 1–21.

[50] Harold N Gabow and Robert Endre Tarjan. 1983. A linear-time algorithm for a special case of disjoint set union. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*. 246–251.

[51] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik Van Der Kouwe. 2016. TypeSan: Practical type confusion detection. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*. Vienna, Austria.

[52] Michael Hind. 2001. Pointer analysis: Haven't we solved this problem yet?. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 54–61.

[53] Shuang Hu, Baojian Hua, and Yang Wang. 2022. Comprehensiveness, Automation and Lifecycle: A New Perspective for Rust Security. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 982–991.

[54] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. 2017. Hextype: Efficient detection of type confusion errors for c++. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. Dallas, TX, USA.

[55] Ofek Kirzner and Adam Morrison. 2021. An analysis of speculative type confusion vulnerabilities in the wild. In *Proceedings of the 30th USENIX Security Symposium (Security)*. Virtual Event.

[56] Stephen C Kleene and Emil L Post. 1954. The upper semi-lattice of degrees of recursive unsolvability. *Annals of mathematics* (1954), 379–407.

[57] Donald E Knuth. 1964. Backus normal form vs. backus naur form. *Commun. ACM* 7, 12 (1964), 735–736.

[58] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2020. Spectre attacks: Exploiting speculative execution. *Commun. ACM* 63, 7 (2020), 93–101.

[59] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. 2015. Type casting verification: Stopping an emerging attack vector. In *Proceedings of the 24th USENIX Security Symposium (Security)*. Washington, DC, USA.

[60] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. 2021. MirChecker: detecting bugs in Rust programs via static analysis. In *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS)*. Virtual Event, Korea.

[61] Jiun Min, Dongyeon Yu, Seongyun Jeong, Dokyung Song, and Yuseok Jeon. 2024. ERASAN: Efficient Rust Address Sanitizer. In *Proceedings of the 45th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.

[62] Anders Møller and Michael I Schwartzbach. 2012. Static program analysis. *Notes. Feb* (2012).

[63] Flemming Nielson, Hanne R Nielson, and Chris Hankin. 2015. *Principles of program analysis*. springer.

[64] Lili Sun, Chenggang Wu, Zhe Wang, Yan Kang, and Bowen Tang. 2022. 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)*. IEEE, 757–766.

[65] Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael R Lyu. 2021. Memory-safety challenge considered solved? An in-depth study with all Rust CVEs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–25.