SafeDrop: Detecting Memory Deallocation Bugs of Rust Programs via Static Data-Flow Analysis

MOHAN CUI, School of Computer Science, Fudan University, China
CHENGJUN CHEN*, School of Computer Science, Fudan University, China
HUI XU†, School of Computer Science, Fudan University, China
YANGFAN ZHOU, School of Computer Science, Fudan University, China and Shanghai Key Laboratory of Intelligent Information Processing, China

Rust is an emerging programming language that aims to prevent memory-safety bugs. However, the current design of Rust also brings side effects, which may increase the risk of memory-safety issues. In particular, it employs OBRM (ownership-based resource management) and enforces automatic deallocation of unused resources without using the garbage collector. It may therefore falsely deallocate reclaimed memory and lead to use-after-free or double-free issues. In this paper, we study the problem of invalid memory deallocation and propose *SafeDrop*, a static path-sensitive data-flow analysis approach to detect such bugs. Our approach analyzes each function of a Rust crate iteratively in a flow-sensitive and field-sensitive way. It leverages a modified Tarjan algorithm to achieve scalable path-sensitive analysis and a cache-based strategy for efficient inter-procedural analysis. We have implemented our approach and integrated it into the Rust compiler. Experiment results show that the approach can successfully detect all such bugs in our experiments with a limited number of false positives and incurs a very small overhead compared to the original compilation time.

CCS Concepts: • **Software and its engineering → Software defect analysis**; **Automated static analysis**; • **Security and privacy → Software security engineering**.

Additional Key Words and Phrases: Rust, Data-Flow Analysis, Meet Over Path, Path Sensitivity

1 INTRODUCTION

Rust is an emerging programming language with attractive features for memory-safety protection yet providing efficiency comparable to C/C++. It achieves that goal by dividing the Rust programming language into safe Rust and unsafe Rust. The boundary is an `unsafe` marker that confines the risks of memory-safety issues (*e.g.,* dereferencing raw pointers) within unsafe code only [9]. Rust guarantees the soundness of safe Rust with no risks of introducing memory-safety issues [8]. Many real-world projects start to embrace Rust due to these features, such as Servo [4] and TockOS [27]. Although the feedback on Rust's effectiveness in preventing memory-safety bugs is positive, there still exist a large number of such bugs in real-world projects [14, 34, 42].

*Contributed equally with Mohan Cui to this research.
†Corresponding author.

Authors' addresses: Mohan Cui, School of Computer Science, Fudan University, China, mhcui20@fudan.edu.cn; Chengjun Chen, School of Computer Science, Fudan University, China, cjchen20@fudan.edu.cn; Hui Xu, School of Computer Science, Fudan University, China, xuh@fudan.edu.cn; Yangfan Zhou, School of Computer Science, Fudan University, China and Shanghai Key Laboratory of Intelligent Information Processing, China, zyf@fudan.edu.cn.

1049-331X/2022/6-ART $15.00
https://doi.org/10.1145/3542948

This work studies a specific type of memory-safety bugs found in real-world Rust crates (projects) related to RAII (resource acquisition is initialization). In particular, Rust employs a novel OBRM (ownership-based resource management) model to make each variable have exclusive ownership, which assumes the resource should be allocated when creating an owner and deallocated once its owner goes out of the valid scope. Ideally, this model should be able to prevent dangling pointers [10] and memory leakages [26], even when a program encounters exceptions (panics). However, we observe that many critical bugs of real-world Rust crates are associated with such an automatic deallocation scheme, *e.g.,* it may falsely drop some buffers that are still being used and incur use-after-free bugs (*e.g.,* CVE-2019-16140), or may falsely drop dangling pointers and cause double free (*e.g.,* CVE-2019-16144).

In general, memory deallocation bugs are triggered by unsafe code. Unsafe APIs are necessary for Rust to provide the low-level control and abstraction over implementation details [14]. However, misusing unsafe APIs can invalidate the soundness of the ownership-based resource management system and may cause undefined behaviors. For example, an unsafe API may lead to memory reclaim of shared aliases, and dropping one instance would create dangling pointers for the remaining aliases [17]. Moreover, the interior unsafe [34] in Rust, allowing a function that contains internal unsafe code and can be declared as safe, may have potential memory-safety issues inside. The current Rust compiler has done little regarding the memory safety risks of unsafe code but simply assumes developers should be responsible for employing them. As memory safety is the most important feature promoted by Rust, reducing such risks is exceedingly important if possible.

Although the problem of detecting memory deallocation bugs has been extensively studied in other languages (*e.g.,* [10, 16, 26]), there are essential differences for Rust due to the unique characteristic of Rust language. These characteristics introduce a new problem setting for performing use-after-free and double-free detection. Firstly, Rust introduces a novel boundary between safe code and unsafe code. The boundary ensures that it is impossible to introduce shared mutable aliases without using unsafe code. Such design can help us to narrow down the culprit statements that may introduce shared mutable aliases to several possible types of API calls (*e.g.,* `from_raw_parts()`) and we could leverage the advantage to design an accurate and efficient bug detection algorithm. Secondly, Rust employs an ownership-based resource management scheme by default for all values. The semantics of ownership movement should be considered to design a decent detection algorithm. To our best knowledge, existing data-flow analysis approaches do not consider ownerships in Rust and their transfer among variables. Note that C++ smart pointers (especially `unique_ptr`) also consider ownership, but their ownerships are only confined to the wrapping values in smart pointers, and they are generally used in a small portion of a C++ program. In contrast, Rust applies ownership to all values. Thirdly, Rust employs RAII and automatically reclaims unused values in both normal execution paths and unwinding paths via a uniform statement `drop()` in the MIR (mid-level intermediate representation) level. Since `drop()` is a major culprit that may introduce dangling pointers issues, we could probe these function calls to detect use-after-free and double-free bugs. We believe these characteristics are non-trivial for considering a new bug detection approach for Rust.

To tackle the memory deallocation bug detection problem for Rust, this paper proposes a novel path-sensitive data-flow analysis approach, namely SafeDrop. SafeDrop analyzes each API of a Rust crate iteratively and scrutinizes whether each `drop()` statement in Rust MIR is safe to launch. Since the underlying problem of alias analysis is NP-hard even if it could be decidable [24], we have adopted several subtle designs to improve the scalability while not sacrificing much precision. Firstly, our approach extracts all valuable paths of a function based on a modified Tarjan algorithm [15], which is effective to eliminate redundant paths in cycles with identical alias relationships. For each path, we extract the alias sets in a flow-sensitive manner and analyze the safety of each `drop()` statement accordingly. When encountering function calls, we analyze the callee recursively, and extract the alias relations between its arguments and return value. To avoid duplicated analysis, we cache and reuse the obtained aliasing result of each function.

We have implemented our approach as a query (similar to a pass) of Rust compiler v1.52 and conducted real-world experiments on existing CVEs of such types. Experimental results show that SafeDrop can successfully recall all these bugs with a limited number of false positives. The analysis overhead generally does not exceed 2× in comparison with the original compilation time. We further apply SafeDrop to 50 more Rust crates and find 12 crates with invalid memory deallocation issues previously unknown.

We summarize the contribution of this paper as follows.

- Our paper serves as the first attempt to study memory deallocation bugs of Rust programs related to the side effect of RAII. We systematically discuss the problem and extract several common patterns of such bugs. Although our work focuses on Rust, the problem may also exist in other programming languages with RAII.
- We have designed and implemented a novel path-sensitive data-flow analysis approach for detecting memory deallocation bugs. It employs several careful designs to be scalable while not sacrificing much precision, including a modified Tarjan algorithm for path-sensitive analysis and a cache-based strategy for inter-procedural analysis.
- We have conducted real-world experiments with existing Rust CVEs and verified the effectiveness and efficiency of our approach. Moreover, we find 12 Rust crates involved with invalid memory deallocation issues previously unknown.

2 PRELIMINARIES

This section presents the background of the problem, including the memory management model of Rust, the basics of the Rust compiler, and the borrow checker of Rust as well as its limitations.

2.1 Memory Management of Rust

Rust introduces a novel ownership-based resource management system to manage memory, and this model assumes each variable has exclusive ownership for its allocated memory. Ownership can be borrowed as references in either mutable or immutable manner with several restrictions (lifetime rules). It requires that reference cannot outlive its referent and the mutable reference cannot be aliased. Rust also provides the traditional raw pointers like references without previous requirements. However, any operations of raw pointers that may violate the memory safety are deemed unsafe and need to be wrapped with the `unsafe` marker *e.g.,* dereferencing raw pointers.

In type system, Rust divides types into two mutex kind of trait: `Copy` trait for stack-only data and `Drop` trait [19] for others. These traits are important in memory management and determine the way for generating lvalue from rvalues in assignments, parameter passing, and value returning. If the value has `Copy` trait, Rust will duplicate (copy) it in the stack and the older variable is still usable. Otherwise, it will transfer (move) the ownership, thus the older variable is no longer available.

The memory management idea of Rust shares many similarities with the smart pointers of C++, and it benefits Rust in realizing automatic RAII, *i.e.,* resources are bounded with valid scopes [35]. In particular, each `Drop` variable is associated with a fixed drop scope[1]. When an initialized variable goes out of the drop scope, the destructor will recursively drop its fields in order. Because Rust intentionally manages the ownership of each `Drop` object at each program point, it can free unused resources without garbage collection by automatically running their destructors once the control flow leaves the drop scope.

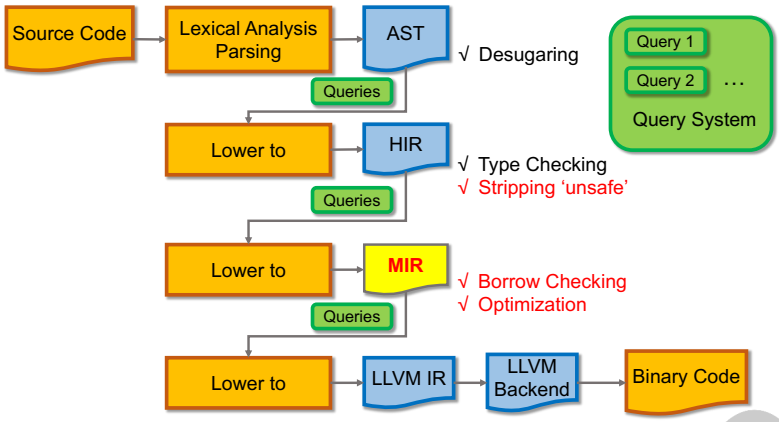


Fig. 1. Procedures of the Rust compiler.

2.2 Basics of Rust Compiler

Rust compiler defines a query system to perform demand-driven compilation, which is a bit different from traditional pass-based systems [37]. Figure 1 presents several main steps for compiling Rust programs, and our key design for detecting memory-safety violations is based on MIR. Compared to LLVM IR (intermediate representation) [25], MIR is a higher-level representation that captures more Rust semantic information.

Listing 1 presents several syntax rules of Rust MIR that are essential for ownership-based resource management, and the full syntax system is available in RFC#1211[2]. According to the syntax, a basic block is composed of several statements and a single terminator. The statements mainly consist of three forms: `Assignment`, `StorageLive`, and `StorageDead`. The `Assignment` assigns the value of `RValue` to `LValue` in different ways. In particular, `move LValue` means transferring the ownership of `LValue` to `RValue`; `& mut LValue` means borrowing `LValue` as a mutable reference; and `* LValue` means creating an immutable raw pointer towards `LValue`. The `StorageLive` and `StorageDead` represent the start and the end of a live range for the storage of a variable respectively. The terminator of each basic block describes how it connects to subsequent blocks. `Drop(Value)` means invoking the destructor of `Value` once `Value` goes out of its drop scope. To achieve automatic resource deallocation for exceptions (a.k.a. panic in Rust), the terminators are generally associated with the exceptions in Rust MIR, *i.e.,* an extra exception handling block `BB1` is added for handling the stack unwinding when invoking function calls. In particular, `SwitchInt()` is a special terminator that we will discuss in Section 4.

2.3 Borrow Checker of Rust

Rust MIR introduces a borrow (reference) checker based on lifetime inference. The lifetime determines whether a reference is valid to use. Non-lexical lifetime (NLL) can save much effort from tediously specifying the lifetime in a fine-grained manner. The mechanism is officially documented in RFC#2094[3], and we highlight the key idea below. NLL extracts a set of constraints based on the liveness, subtyping, and reborrowing requirements. *Liveness constraint* means a reference should be valid from declaration to the last use; *subtyping constraint* means the lifetime of a reference should not exceed its referent; *reborrow constraint* means the lifetime of a reference, that is

[1]https://doc.rust-lang.org/reference/destructors.html
[2]https://rust-lang.github.io/rfcs/1211-mir.html
[3]https://rust-lang.github.io/rfcs/2094-nll.html

Listing 1. Core syntax of Rust MIR.

```
BasicBlock := {Statement} Terminator
Statement := LValue = RValue | StorageLive(Value)
        | StorageDead(Value) | ...
LValue := LValue | LValue.f | *LValue | ...
RValue := LValue | move LValue
        | & LValue | & mut LValue
        | * LValue | * mut LValue | ...
Terminator := Goto(BB) | Panic(BB) | Drop(Value)
        | Return | Resume | Abort
        | If(Value, BB0, BB1)
        | LValue = (FnCall, BB0, BB1)
        | SwitchInt(Value, BB0, BB1, BB2, ...) | ...
```

created from the object by dereferencing other references, should not exceed its referent. Rust compiler then solves the constraints and computes the minimal lifetime of each reference via fixed-point iteration. In this way, it can generate an optimized solution. Thus, the borrow checker can detect memory-safety violations that are infringing the lifetime rules incurred by references.

2.4 Soundness Limitations of Rust Compiler

As shown in Figure 1, the Rust compiler strips `unsafe` markers before lowering the code to MIR. The borrow checker, which is a component in MIR, can be applied to both safe code and unsafe code. This method is valid only for objects and references. The unsafe code with raw pointers can easily breach the safety promise and lead to invalid memory reclaim issues. In other words, the memory model cannot discriminate the alias relationship between multiple `Drop` objects, because it only calculates the fixed drop scope of each object. To our best knowledge, Rust has no such algorithm for ensuring the safety of memory reclaim related to raw pointers, which is the problem studied in our paper.

3 PROBLEM STATEMENT

3.1 Motivating Example

Rust enforces RAII and releases unused resources automatically. In practice, this mechanism may falsely deallocate some buffers and is prone to trigger memory-safety issues. Based on the occurrence of a buggy `drop()` statement, we categorize such problems into two situations: invalid drop in normal execution path and invalid drop in exception handling path.

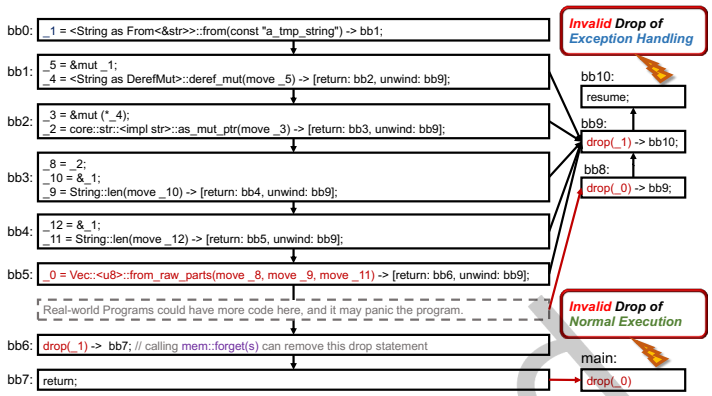
3.1.1 ***Invalid Drop of Normal Execution.*** Such a bug happens if the culpriting `drop()` statement locates in a normal execution path, and the parameter of `drop()` is not safe to launch. We use Figure 2a as a proof-of-concept (PoC) example to demonstrate such cases.

The function `genvec()` is an interior unsafe constructor that creates vector `v` from a raw pointer `ptr`. In this PoC, the string `s` and the vector `v` are sharing the same heap space. After the automatic deallocation of the string, the vector will be a dangling pointer to the released buffer, and using this vector later in the `main()` function will incur use after free. As long as the vector is reaching the end of its drop scope, the automated deallcation of the vector will incur double free. In Figure 2b, the returned vector `_0` in `bb5` is created based on `_1` with an alias propagation chain `1->5->4->3->2->8->0`. Therefore, `_0` contains an alias pointer of `_1`. Namely, dropping `_1` incurs dangling pointers of `_0` and dropping `_0` later in the `main()` function incurs double free. There are

```rust
fn genvec() -> Vec<u8> {
    let mut s = String::from("a_tmp_string");
    let ptr = s.as_mut_ptr();
    let v;
    unsafe{
        v = Vec::from_raw_parts(
                ptr, s.len(), s.len());
    }
    // mem::forget(s); // do not drop s
    // otherwise, s is dropped before return
    return v;
}

fn main() {
    let v = genvec();
    // use v -> use after free
    // drop v before return -> double free
}
```

(a) Source code of dropping aliases.

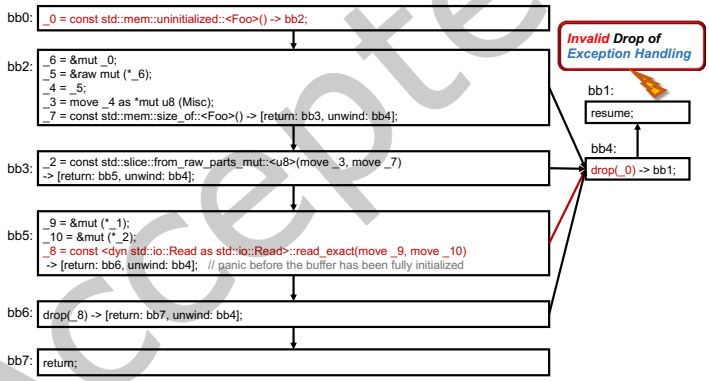


(b) MIR form and CFG of Figure 2a.

```rust
struct Foo{
    vec : Vec<i32>,
}

impl Foo {
    pub unsafe fn read_from(src: &mut Read) -> Foo{
        let mut foo = mem::uninitialized::<Foo>();
        let s = slice::from_raw_parts_mut(
                &mut foo as *mut _ as *mut u8,
                mem::size_of::<Foo>());
        src.read_exact(s);
        foo
    }
}
```

(c) Source code of dropping uninitialized memory.



(d) MIR form and CFG of Figure 2c.

Fig. 2. Motivating example of memory-safety issues incurred by automatic deallocation.

many such bugs found in practice, such as CVE-2019-16140, which is use-after-free, and CVE-2018-20996 and CVE-2019-16144, which are double free.

It seems hard for the borrow checker to analyze raw pointers to detect such issues. The problem lies in the trade-off design for dealing with function calls. For example, `_4` is created by calling `deref_mut(move _5)`, and its parameter `_5` is a mutable alias of `_1`. For simplicity, Rust assumes that each parameter should either transfer its ownership to the callee for the `Drop` variable or duplicate a deep copy for the `Copy` variable, and the return value would no longer share the ownership with the alive variables remained. Such assumptions save the borrow checker from the inter-procedural pointer analysis. As a trade-off, it leaves the holes for raw pointers in unsafe Rust. Note that `from_raw_parts()` is an unsafe function, and it is the culprit that leads to the memory reclaim in this example. However, the `unsafe` marker has been stripped away after lowering the code to MIR, and the Rust compiler cannot differentiate the safety boundary anymore. In conclusion, the current Rust compiler does not support checking alias for raw pointers. Therefore, the problem cannot be fixed easily in the current framework.

3.1.2 ***Invalid Drop of Exception Handling.*** In some real-world bugs, memory deallocation issues only exist in the exception handling path, as the program panics and enters into the unwinding process. For example,

Listing 2. Typical patterns checked for invalid memory deallocation in SafeDrop. (UAF: use after free; DF: double free; IMA: invalid memory access)

```
Pattern 1: getPtr() -> unsafeConstruct() -> drop() -> use()  => UAF
Pattern 2: getPtr() -> unsafeConstruct() -> drop() -> drop() => DF
Pattern 3: getPtr() -> drop() -> unsafeConstruct() -> use()  => UAF
Pattern 4: getPtr() -> drop() -> unsafeConstruct() -> Drop() => DF
Pattern 5: getPtr() -> drop() -> use() => UAF
Pattern 6: uninitialized() -> use()  => IMA
Pattern 7: uninitialized() -> drop() => IMA
```

CVE-2019-16880 and CVE-2019-16881 have double-free issues if the program panics; CVE-2019-15552 and CVE-2019-15553 may drop uninitialized memory in exception handling. Figure 2 presents this PoC to demonstrate the problem.

Suppose developers have fixed the bug by adding `mem::forget()` to the code in Figure 2a which prevents `drop(_1)` in `bb6` in Figure 2b. There may also have some statements (*e.g.,* retrieving the content of the vector) between creating `v` and calling `mem::forget()`. In this way, these additional statements are still vulnerable program points. If the program panics in these statements, according to the principle of RAII, Rust should deallocate resources during stack unwinding by continuously calling `drop()`. Considering the aliasing `Drop` objects, deallocating them would incur double-free issues. Moreover, `mem::forget()` will consume the owner of the string, it should be used as late as possible to make the string usable.

Likewise, dropping uninitialized memory is another popular issue during exception handling. Figure 2c and 2d demonstrate a PoC that applies an uninitialized buffer first and initializes it afterwards. However, if the program panics before the buffer has been fully initialized, the stack unwinding process would deallocate those uninitialized memories, which is similar to the use-after-free issue if the buffer has pointers.

3.2 Problem Definition

We formalize the problem as follows. Supposing a programming language with RAII can deallocate unused memory automatically based on the static strategies. Due to the limitations of static analysis, such deallocations may cause memory-safety bugs. In particular, there are two types of invalid memory deallocations.

Definition 3.1 (Drop buffers in use)**.** If the algorithm falsely deallocates some buffers that will be accessed later, it would incur dangling pointers that are vulnerable to memory-safety issues, including use after free and double free.

Definition 3.2 (Drop invalid pointers)**.** If the invalid pointer is dangling, dropping the pointer would incur double free; if the invalid pointer points to an uninitialized memory that contains other pointers, dropping the pointer may deallocate the nested pointers recursively and incur invalid memory access.

The problem of invalid deallocation should be a common issue for programming languages that enforce RAII, such as C++ and Rust. However, it is more severe in Rust due to Rust emphasizing much more on memory safety, and such security issues are less tolerable.

3.3 Typical Patterns

We illustrate several typical patterns of such bugs. Note that since the data-flow approaches for normal execution paths and exception handling paths are similar, we do not further differentiate them in these patterns. Listing 2 summarizes 7 typical patterns of invalid memory deallocation from the view of statement sequences.

Pattern 1 and pattern 2 correspond to our PoC of Figure 2b that incurs use after free (UAF) or double free (DF). We employ `getPtr()` as a general representation for obtaining a raw pointer to a `Drop` object, and employ `unsafeConstruct()` to denote unsafe functions for constructing new aliasing objects (*e.g.,* `from_raw_part()`). Pattern 3 and pattern 4 are similar but switch the order of `unsafeConstruct()` and `drop()`. Pattern 5 has no `unsafeConstruct()` but uses the dangling pointer directly and incurs use after free. Pattern 6 and pattern 7 correspond to our PoC of Figure 2d that relates to uninitialized memory. We employ `uninitialized()` to represent constructing uninitialized `Drop` object. Either using uninitialized memory or dropping it directly is invalid memory access (IMA).

According to the summarized vulnerability patterns, we can observe that all patterns contain at least an unsafe constructor or `drop()`. Informally, an unsafe constructor generally inputs a pointer and returns an object owning the pointed memory. For example, `from_raw_parts()` is a commonly used unsafe constructor supported by many containers in Rust standard libraries, such as `String`, `Vec`, *etc.* Such constructors are essential for these containers to create an object owner based on a pointer input. `uninitialized()` can also be viewed as a special unsafe constructor that returns an object with uninitialized memory. Different from C/C++, Rust employs `drop()` as the uniform destruction API for all `Drop` objects. Therefore, we can leverage such characteristics to simplify our detection algorithm design because such APIs can serve as important clues for searching memory deallocation issues among alias sets. Besides, we do not need to maintain the alias sets involving no unsafe constructor or `drop()`.

3.4 Research Challenge

Traditional bug detection approaches include dynamic analysis and static analysis. Dynamic analysis is less efficient for this problem because the condition required for triggering an unwinding path cannot be satisfied easily. For example, there are rigorous conditions required for panicking the program of Figure 2d at a particular program point (*e.g.,* `bb2`, `bb3`, or `bb5`). Existing dynamic analysis approaches such as fuzzing can hardly generate test cases to cover these unwinding paths. Symbolic execution also suffers critical limitations because such conditions generally involve difficult memory modeling issues for path constraint extraction [43]. Therefore, we prefer static analysis for this problem over dynamic analysis.

Static analysis also faces several challenges in solving the problem. One major challenge lies in interpreting the semantics of Rust MIR related to its novel type system, such as the features of ownership, mutability, and trait. A static analysis algorithm should carefully deal with the alias relationships incurred by different Rust MIR instruction forms, such as move, mutable borrow, immutable borrow, and dereference. Besides, the feature of a Rust object depends on the traits implemented by the object type. Only the type with the `Drop` trait will be deallocated by calling its destructor when the lifetime ends. For a compound type, such traits can be automatically derived according to the types of its subfields. Therefore, we should infer the traits of each type to better capture MIR semantics. Due to such uniqueness of Rust MIR, existing static analysis approaches are not directly applicable to the problem.

Finally, such a static analysis tool should demonstrate good usability without many false positives. However, the underlying problem involves alias analysis which is NP-hard. It is challenging for the proposed static analysis approach to be both efficient and precise.

4 APPROACH

In this section, we describe our approach for detecting invalid memory deallocation problems described in Section 3.

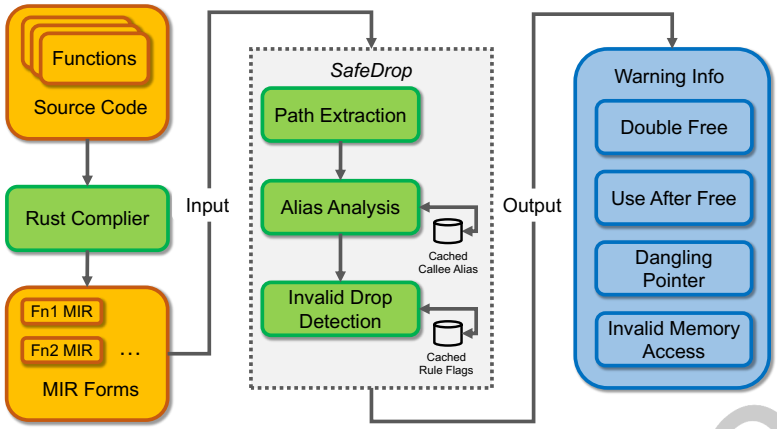


Fig. 3. Overall framework of SafeDrop.

4.1 Overall Framework

We tackle the problem with a path-sensitive data-flow analysis approach integrated into the Rust compiler, namely SafeDrop. Figure 3 overviews the framework of our approach. It inputs the MIR of each function and outputs warnings of potential invalid memory deallocation issues along with the buggy code snippets. There are three essential steps in the framework highlighted as follows.

- ***Path Extraction:*** SafeDrop adopts a path-sensitive approach or meet-over-paths (MOP [2]). Since the number of paths for one function could be very large or even infinite, we employ a novel method based on the Tarjan algorithm to merge redundant paths and generate a spanning tree. This step traverses this tree and finally enumerates all valuable paths.
- ***Alias Analysis:*** SafeDrop performs flow-sensitive and field-sensitive alias analysis for each path to build alias sets among variables or their subfields for composite types. Meanwhile, the analysis is inter-procedural but context-insensitive by caching and reusing the alias sets among the function arguments and the return value.
- ***Invalid Drop Detection:*** Based on the alias sets established before, this step searches vulnerable drop patterns for each data flow on the path and extracts corresponding suspicious code snippets.

Next, we elaborate on each of these steps in more detail.

4.2 Path Extraction

SafeDrop adopts a path-sensitive method to achieve precise results rather than using the traditional semi-lattice scheme [20]. It traverses the control-flow graph of a function and enumerates all target paths. In our approach, a target path has two criteria: 1) a unique set of code blocks with a start node (entrance) and an exit node (return), and 2) the set of a valuable path should not be the subset of any other valuable paths. If two paths are conflicting due to the second criterion, we only keep one with more unique blocks because it should be more vulnerable [3]. In this way, we do not need to visit each cycle repeatedly on the control-flow graph but only consider the maximum set of the blocks in a strongly connected component. Besides, we do not need to differentiate unwinding paths from normal execution paths, and our algorithm works the same on both of them.

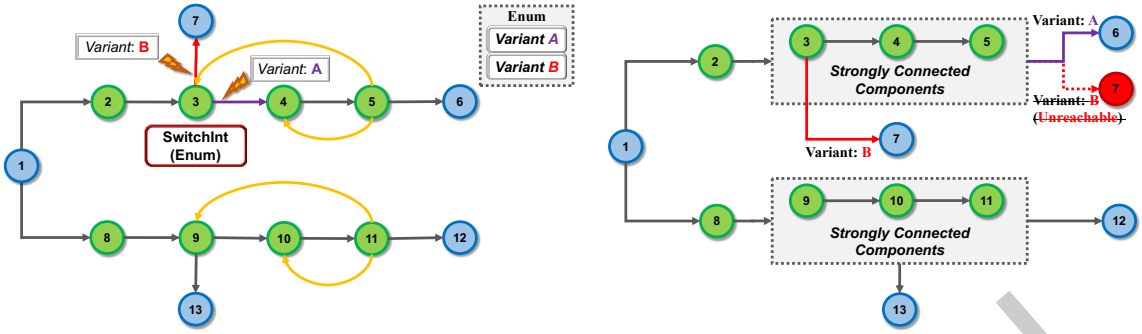
Algorithm 1 demonstrates our main algorithm for analyzing each Rust function. Overall, the algorithm firstly uses a modified Tarjan algorithm [15] to generate a spanning tree of the graph (Line 1-6) and then traverses the

Algorithm 1: Algorithm for analyzing each function of a Rust crate.

```
Data: cfg: control-flow graph of the function being analyzed ;
talias: alias sets and their taint labels ;
fnalias: alias information for inter-procedural analysis ;
bug: detected bugs;

1  bug ← false;
2  sccs ← Tarjan(cfg);                          // extract all strongly connected components
3  st ← GenerateTree(cfg, sccs);                // derive the spanning tree
4  foreach node in st do
5      if st.ContainsConflict(node) then
6          st ← ResolveConflict(st, node);
7  DFSTraverse(talias, fnalias, bug, st.root);
8  PrintBugWarning(bug);                        // output the warnings
9  Cache(fnalias);                              // for reuse when analyzing other functions
   /* recursive traversal over the spanning tree                                        */
10 Function DFSTraverse(talias, fnalias, bug, node)
11     foreach statememt in node do
12         AnalyzeAlias(talias, node);          // Section 4.3
13         DetectBug(talias, node);             // Section 4.4
14         children ← GetChildren(node);
15         if children.IsNone() then
16             Merge(fnalias, bug);             /* merge the results from different paths */
17             return
18     foreach child in children do
19         DFSTraverse(talias, fnalias, bug, child)
```

spanning tree to obtain and analyze all valuable paths (Line 7). In particular, Line 2 uses the traditional Tarjan algorithm to detect the strongly connected components (SCC) of a graph. Line 3 then shrinks each strongly connected component into a point and generates a spanning tree for the graph. Ideally, the spanning tree should explicitly represent all valuable paths by enumerating each route from the tree root to each leaf node. However, such a coarse-grained approach may lead to mistakes due to some specific Rust statements. For example, Rust introduces a critical design for the enumeration types that inaccurately make the traditional Tarjan algorithm. Figure 7a presents an example with two mutually exclusive paths that should not be shrunk into one point, *i.e.,* node 3 to node 7 and node 3 to node 4. Because the parameter of `SwitchInt()` is an enumeration type whose value exclusively determines the control flow, the two paths cannot be reachable at the same time. To address the issue, we pinpoint successive `SwitchInt()` statements for the same enumeration parameter and enumerate valid paths for each variant once (Line 4-6). In this way, we can avoid the invalid path `1-2-3-4-5-7` in Figure 4b and generate the valid path `1-2-3-7` only.



(a) Control-flow graph example with SwitchInt(). (b) Spanning tree generated for Figure 4a.

Fig. 4. A sample control-flow graph with SwitchInt() and the corresponding spanning tree. The SwitchInt() of node 3 dispatches the control flow based on the instantiated type of an enumeration type, which could either be variant A or B. The spanning tree should not contain a path that requires the type to be both A and B.

Listing 3. Type filter for alias analysis. The compound types will be filtered if and only if all their fields can be filtered recursively.

```
Value := Type::{Bool, Char, Int, UInt, Float}:  => discard
    | := Type::Array        => depends on the element type
    | := Type::Tuple        => depends on the type of each item
    | := Type::Structure    => depends on each field
    | := Type::Enumeration  => depends on each variant
    | := Type::Union        => depends on each field
```

Listing 4. Types of statements leading to alias relationships. S_a and S_b represent the alias sets of variables a and b respectively; Rule three and four are approximated by ignoring pointer levels; Rule five and six update the alias sets based on the result of the inter-procedural analysis. Note that we will remap the alias set of variable a from S_a to S_a after applying rules 1-4.

```
LValue := Use::Move(RValue)     : e.g., a = move b      =>  S_a = S_a − a,  S_b = S_b ∪ a
    |  := Use::Copy(RValue)     : e.g., a = b           =>  S_a = S_a − a,  S_b = S_b ∪ a
    |  := Ref/AddressOf(RValue) : e.g., a = &b          =>  S_a = S_a − a,  S_b = S_b ∪ a
    |  := Deref(RValue)         : e.g., a = *(b)        =>  S_a = S_a − a,  S_b = S_b ∪ a
    |  := Fn(Move(RValue))      : e.g., a = Fn(move b)  =>  Update(S_a, S_b)
    |  := Fn(Copy(RValue))      : e.g., a = Fn(b)       =>  Update(S_a, S_b)
```

4.3 Alias Analysis

Now we discuss our approach for interpreting Rust MIR and generating alias sets for each path. We first simplify our discussion for intra-procedural alias analysis and then extends it to inter-procedural analysis.

4.3.1 ***Intra-Procedural Alias Analysis.*** At the MIR level, the Rust type system enforces that only the object owner implementing the `Drop` trait will be dropped at the end by calling the uniform destruction function `drop()`.

Listing 5. Sample MIR code and corresponding alias sets generated by flow-sensitive alias analysis.

```
Statement 1:  _2 = &_1;          // alias set:{_1, _2}
Statement 2:  _1 = move _4;      // alias sets:{_1, _4}, {_2}
Statement 3:  _3 = &_1;          // alias sets:{_1, _3, _4}, {_2}
```

Therefore, our analysis is mainly interested in tracing the `Drop` objects and their aliases. In general, an object is created via the assignment expression `LValue = RValue`. Intuitively, we can start our analysis by checking if a newly created object is `Drop` and trace its aliases. However, this would miss the aliases of the objects established with the pointer types. Thanks to another principle of Rust that a type cannot implement both `Copy` trait and `Drop` trait at the same time, we can therefore filter irrelevant objects of `Copy` trait except they contain raw pointers, references, or slices. Listing 3 summarizes several irrelevant types that can be filtered. This is straightforward for primitive types, *e.g.,* all integer and floating-point types implementing `Copy` trait and should be filtered. For composite types of `Copy` trait, we should filter the type by recursively checking if all its subfields can be filtered.

After applying the filter, we employ flow-sensitive analysis to each statement of a path iteratively and extract the alias relationship. We use union-find disjoint sets to store the alias relationship. For each statement introducing new aliases, we update corresponding alias sets and merged two sets automatically if an item in set S_a is aliasing to another item in set S_b.

Based on the semantics of Rust MIR, we can summarize six types of statements that contribute to alias relationships as shown in Listing 4. Type one moves the ownership of `RValue` to `LValue`, and type two uses an `RValue` of `Copy` trait. Although the two statements have different semantics, they incur the same changes to the alias sets. Similarly, our aliasing rules do not need to differentiate mutable and immutable aliases. Rule three and four are addressing and dereference operations correspondingly. Because the bug patterns of Listing 2 mostly hold for aliases of different levels, we can simply approximate their introduced alias relationships by ignoring the levels of pointers. The sole exception is only when the object of a compound type contains two disjoint aliases. Such false positives can be avoided via field-sensitive analysis discussed later in Section 4.3.3. Type five and six are special cases of the first two types with function calls. For such statements, we update the alias sets based on the result of the inter-procedural analysis. The details will be discussed in Section 4.3.2.

Listing 5 demonstrates how we apply the aliasing rules. The first statement leads to an alias set `{_1, _2}`. Then the second statement removes `_1` from the set and establishes a new alias set `{_1, _4}`. The third statement adds `_3` to the alias set of `{_1}`. Finally, the path has two alias sets `{_1, _3, _4}` and `{_2}`. Note that flow-sensitive analysis is essential because we may falsely think `_3` is an alias of `_2` if not considering the happened-before relationship of the first two statements.

4.3.2 ***Inter-Procedural Analysis.*** When meeting function calls, we perform inter-procedural analysis to obtain the alias relationship among functional parameters and the return value. In general, our analysis is context-insensitive and does not consider how a function is invoked. Therefore, we can simply merge all possible alias relationships incurred in all paths as the aliasing rule of the specific function and use the rule to update the alias sets of the caller. For each path of the callee function, the aliasing rule is the same as we have discussed previously in the intra-procedural analysis.

The analysis of each callee function is performed recursively until all the involved functions have been analyzed. We can cache the analysis result of each function into the hash table for further reuse. When encountering recursive loops, we adopt a fixed-point iteration method [1] to solve the problem. In other words, we assume an initial alias set (S_0) for the recursive function, and iteratively update the alias set $S_1, S_2, .., S_{n+1}$ using the previous result until S_n is no longer changed.

Listing 6. Sample MIR code for field-sensitive inter-procedural analysis.

```
enum E { A, B { ptr: *mut u8 } }
struct S { b: E }

fn foo(_1: &mut String) -> S:
    _3 = str::as_mut_ptr(_1);  // alias set: {_3, _1}
    ((_2 as B).0: *mut u8) = move _3;   // alias set: {_2.0, _3, _1}
    discriminant(_2) = 1;  // instantiate the enum type to variant B
    (_0.0: E) = move _2; // alias sets: {_0.0, _2}, {_0.0.0, _2.0, _3, _1}
    return;

fn main():
    _1 = String::from("string"); // alias set: {_1},
    _2 = &mut _1;   // alias set: {_2, _1},
    _3 = foo(move _2); // alias set: {_3.0.0, _2, _1}
    ...
```

4.3.3 ***Field Sensitivity***. For composite types with multiple fields, we employ a field-sensitive approach to achieve better precision. Rust MIR represents the identifier of each subfield explicitly with the index of the subfield. It is convenient for us to record the subfields in a linear form with any field depth. For example, `_0.0` means the first field of the variable `_0`. Therefore, we can integrate field-sensitive analysis seamlessly into our previously discussed aliasing rules except for one extra extension, *i.e.,* we should update the alias sets that contain the descendant fields of the newly generated aliases. Taking Figure 6 as an example, the last statement of the `foo()` function before `return` lead to a new alias set `{_0.0, _2}`. Since another alias set `{_2.0, _3, _1}` contains one descendent field of `{_2}`, we should also update the set to `{_0.0, _2.0, _3, _1}`. In this way, the alias set of the `main()` function after executing `foo()` should be `{_3.0.0, _2, _1}` instead of `{_3, _2, _1}`.

More generally, we can assume a struct has multiple fields of pointers, such as `_2.0` and `_2.1`. In this case, our algorithm will maintain two separate alias sets for `_2.0` `_2.1` respectively. Once the alias set of `_2` is updated, we will automatically synchronize the changes to the alias sets of `_2.0` and `_2.1`. However, if the alias set of a descendent field is changed, we do not update the parent set. Since our algorithm maintains an alias set for each subfield, it would serve as an important complement to our approximation of pointer levels presented in Section 4.3.1.

4.4 Rules for Bug Detection

Based on the established alias relationships, we can examine whether a path involves memory-safety issues. According to the bug patterns discussed in Section 3.3, a buggy path generally involves a `drop()` statement or the special unsafe constructor `uninitialized()`. Therefore, we can label the alias sets of such objects as the tainted source and propagate the taint information according to the established alias sets. For composite types, we should label each subfield as a taint source. For the taint source generated from `uninitialized()`, we should update the tainted sets as untainted once the object has been initialized.

Based on such taint information, we can summarize the bugs patterns into four rules for detecting them.

- ***Use After Free:*** A variable being used in a statement is tainted.
- ***Double Free:*** A variable being deallocated by `drop()` is tainted.
- ***Invalid Memory Access:*** A variable being used or deallocated is tainted as uninitialized.

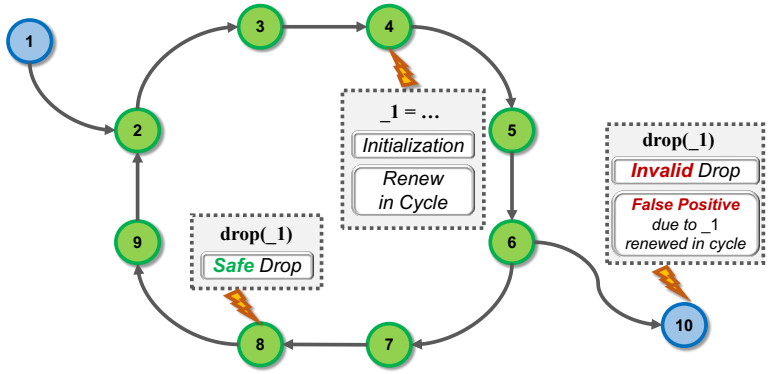


Fig. 5. A corner case of renewing after dropping in a cycle. The refinement of such issue regards the deallocation in node 10 as a safe deallocation in the path 1-(2-3-4-5-6-7-8-9)-10.

- ***Dangling Pointer:*** A variable being returned by one function is tainted. Although this is unsafe only if we use the variable later, this rule is useful to detect deallocation bugs.

Finally, Algorithm 1 integrates all these discussed procedures into one function, `DFSTraverse()`. It performs alias analysis and bug detection while recursively traversing the generated spanning tree.

4.5 Corner Case Handling

The modified Tarjan algorithm generates a spanning tree to extract a valuable path that removes the redundant traversal over cycles. There still exists one kind of corner case that may introduce false positives. For instance, `_1` of Figure 5 is initialized and then deallocated in the SCC `2-3-4-5-6-7-8-9`. The deallocation operation of node `10` should be safe considering the path `1-2-3-4-5-6-10` or `1-2-3-4-5-6-7-8-9-2-3-4-5-6-10`. However, node `10` will treat it as an invalid deallocation using default SCC-based method, *i.e.,* `1-(2-3-4-5-6-7-8-9)-10`. This is a false positive because the deallocated `_1` can be renewed in the node `4` before reaching node `10`. To deal with such cases, we have refined our approach by further checking whether the path range between two deallocations for the same variable is separated by a definition of the variable. In this way, we can detect and avoid such false warnings afterward.

5 EVALUATION

5.1 Implementation

We have implemented our approach on top of Rust MIR and integrated it into the Rust compiler as a query. MIR is a desugared form of Rust code, which is sufficient for our analysis task. The query can either be invoked automatically after running all optimization passes in MIR or can be used when needed. We have integrated the query into the Rust compiler v1.52, and it can be used by the command-line tools, including both `rustc` and `cargo`.

5.2 Experimental Setup

Next, we study the performance of our approach. Specifically, we are interested in the following four questions.

RQ1: How does SafeDrop perform in detecting memory deallocation CVEs concerning recall and precision?

Table 1. Experimental result with existing CVEs. For each host crate of the CVEs, we present the numbers of particular warning types reported by SafeDrop in true positive and false positive (TP/FP). We also apply Rudra, MirChecker, and Miri to these crates. Since Miri is a dynamic analysis tool, we also present the number of test cases for each crate. Two results of Miri are labeled as N.A. because it does not support foreign functions. LoC: lines of code of the crate.

Crate			**CVE**		**SafeDrop Report (TP/FP)**						**Rudra**	**MirChecker**	**Miri**	
Name	# Methods	LoC	CVE-ID	Type	UAF	DF	DP	IMA	Total	Recall	TP/FP	TP/FP	# Tests	TP/FP
isahc	89	1304	2019-16140	UAF	-	0/1	1/0	-	1/1	100%	0/0	0/0	6	0/0
open-ssl	1188	20764	2018-20997	UAF	1/2	-	0/1	-	1/3	100%	0/0	0/0	0	N.A.
linea	1810	24317	2019-16880	DF	-	1/0	-	10/0	11/0	100%	1/2	0/5	2	0/0
ordnung	145	2546	2020-35891	DF	0/1	-	3/0	-	3/1	100%	1/3	0/2	31	0/0
crossbeam	221	4184	2018-20996	IMA	-	0/1	-	2/0	2/1	100%	0/0	0/0	21	0/1
generator*	158	2608	2019-16144	IMA	-	-	-	1/0	1/0	100%	0/0	0/0	0	N.A.
linkedhashmap	137	1974	2020-25573	IMA	-	-	-	1/0	1/0	100%	0/0	0/0	39	0/1
smallvec*	187	2297	2018-20991 2019-15551	DF DF	-	-	1/2	1/0	2/2	100%	2/1	0/2	48	1/1

RQ2: How much overhead does SafeDrop introduce compared to the original compilation process?
RQ3: Does SafeDrop perform better than other existing approaches, such as Rudra [7] and MirChecker [28]?
RQ4: Is SafeDrop useful for helping developers in detecting unknown bugs from potentially many alarms?

To answer RQ1, we have collected a dataset of related CVEs for evaluation [42], including two use-after-free issues, three double-free issues, and four invalid memory access issues. These nine CVEs are from eight Rust crates. Meanwhile, we measure the analysis time of SafeDrop and calculate the overhead compared with the original compiler to answer RQ2. To answer RQ3, we also apply, Rudra [7], MirChecker [28] and Miri [31] to these eight creates. Finally, we choose 50 more real-world Rust crates from GitHub for the usability study of RQ4. We conduct these experiments on a 2.00 GHz Intel processor with the 18.04.1-Ubuntu operating system (Linux kernel version 5.4).

5.3 Results and Analysis

5.3.1 ***Effectiveness.*** We apply SafeDrop to the crates of the reported CVEs to examine whether our approach can recall or locate these bugs. Table 1 lists all related CVEs as well as our experimental results. For each crate, SafeDrop reports the detected bugs based on the rules in Section 4.4 and we manually check the correctness of each report.

In general, SafeDrop can recall all these bugs with a small number of false positives. Note that a CVE may involve several buggy code snippets. Therefore the number of reported warnings is a bit larger than the number recorded in CVEs. Besides, the bug types reported by SafeDrop may differ from the original CVE type mainly because the dangling pointer is not a CVE type but should be a problem by our rules. Specifically, returning a dangling pointer from the callee may cause either use-after-free or double-free issues depending on the operation of the caller. The experiment result also demonstrates the scalability of SafeDrop. Table 1 contains crates of different scales in lines of code. SafeDrop is effective on all of them and does not incur more false positives for the programs with more lines of code.

5.3.2 ***Efficiency.*** SafeDrop is a query integrated into the Rust compiler. As a component of the compiler, it should not incur much compilation overhead. Therefore, we evaluate the overhead of SafeDrop by comparing it with the original compilation.

We run each experiment five times and report the average cost. Figure 6 shows the experiment overhead over 8 CVE crates in Table 1. In conclusion, the overhead of SafeDrop generally does not exceed 2× in comparison with the original compilation. The overhead for crate `ordnung` is more obvious because it contains a large amount of

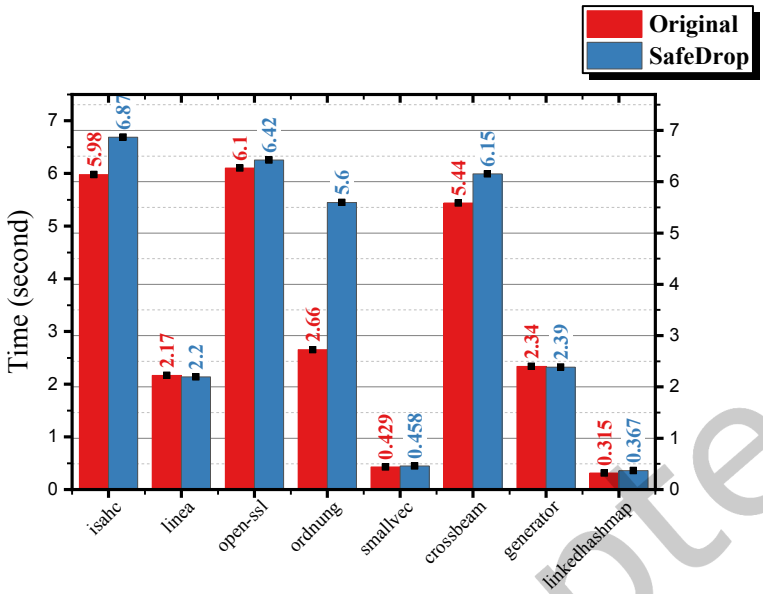


Fig. 6. Experimental result on the efficiency of SafeDrop. We compare the time spent by SafeDrop (including compilation time) with the original compilation time for each crate.

successive `SwitchInt()` which increases the analysis complexity. Moreover, the results also demonstrate that the overhead is irrelevant to the program size. For example, the overhead of `open-ssl` and `smallvec` differs slightly, although `open-ssl` is ninefold larger in size.

5.3.3 ***Comparison with Other Tools.*** Now, we compare SafeDrop with existing tools that can also detect memory deallocation bugs for Rust programs, including `Rudra` [7], MirChecker [28], and `Miri` [31]. The result is shown in Table 1.

Rudra is a static analysis approach that can detect memory-safety bugs based on predefined code patterns. We apply Rudra to the crates of Table 1 and analyze the bugs reported by Rudra. Our result shows that Rudra can detect four bugs out of nine existing CVEs. In particular, it can successfully detect all unwinding bugs related to specific APIs, such as CVE-2019-16880 caused by `uninitialized()` and CVE-2018-20991 related to `write()`. However, it cannot detect other unwinding bugs, such as CVE-2019-16140 related to `Vec::from_raw_parts()`, and memory deallocation bugs in normal execution paths. The main reason is that Rudra adopts an ad hoc approach based on code patterns and does not perform fine-grained data-flow analysis. While Rudra supports more bug patterns, our approach should be more precise than Rudra in detecting memory deallocation bugs.

MirChecker is also a static analysis dedicated to Rust program bug detection. It can detect both run-time crashes (named *run-time panics*) and memory-safety errors (named *lifetime corruption*) by tracing numerical and symbolic information. For a fair comparison, we use the default scripts provided by MirChecker and collect the output related to the *lifetime corruption* issue. Our result in Table 1 shows that MirChecker cannot detect the CVEs listed in our paper. We have double-checked the result with the author of MirChecker. We think the main reason lies in the insufficient support for inter-procedural analysis, and it only analyzes the local crate functions. Furthermore, MirChecker cannot detect memory deallocation bugs in unwinding paths because it mainly analyzes unexpected program panic during normal execution.

Table 2. Experimental results with more crates that involve unsafe constructors. SafeDrop has successfully detected new bugs from 12 creates. Some results are labeled as N.A. due to timeout or support issues. In the marked crates, the unsafe constructors are lying in the extern functions. Our experiments transform them into normal functions and construct calling sequences to simulate C calls.

Crate			**SafeDrop Report (TP/FP)**					**Rudra**	**MirChecker**	**Miri**
Name	# Methods	Loc	UAF	DF	DP	IMA	Total	TP/FP	TP/FP	TP/FP
stretch*	4280	78350	-	24/2	-	-	24/2	N.A.	0/0	N.A.
idroid*	5484	73856	-	7/0	-	-	7/0	N.A.	0/0	N.A.
rose-tools*	996	19160	-	27/3	-	-	27/3	N.A.	0/0	N.A.
wasm-gb	165	5719	-	-	20/0	-	20/0	N.A.	0/0	0/0
rust-poker	113	2298	-	1/0	-	-	1/0	0/0	0/0	N.A.
teardown-tree	677	7258	-	0/2	1/0	-	1/2	0/3	0/4	N.A.
apres-bindings	18	1139	-	-	14/0	-	14/0	0/0	0/0	0/0
rust-workshop	13	1897	-	-	2/0	-	2/0	0/0	0/0	0/0
teraflops	40	606	-	-	2/0	-	2/0	0/0	0/3	0/0
rust-libcint	37	600	-	2/0	-	-	2/0	0/0	0/0	0/0
bzip2	20	170	-	-	2/1	-	2/1	0/0	0/0	0/0
rust-webassembly	12	118	-	-	1/0	-	1/0	0/0	0/1	0/0

Miri is a dynamic analysis tool that can interpret Rust MIR being executed. Therefore, the result of Miri largely depends on adopted test cases. We apply Miri to the crates directly with their unit tests to avoid bias. Our result in Table 1 shows that Miri can only detect one bug of invalid memory access, CVE-2018-20991. We think the main reason is that the default test cases cannot trigger other bugs, which is the limitation faced by dynamic analysis. Achieving better testing coverage and triggering these bugs is another problem orthogonal to Miri. The problem is very challenging for testing Rust libraries, as well as covering unwinding paths. Addressing the problem requires synthesizing programs from library APIs [41], and fuzzing or symbolic executing of these programs. In short, SafeDrop should be more effective than Miri in detecting memory deallocation bugs.

In short, our work focuses on a specific problem related to bad memory deallocation, and we show our approach performs better than other existing tools on this particular problem. However, our work is incapable of detecting other types of bugs reported by other tools. Therefore, the benchmarks employed by other tools are not directly applicable to our evaluation. In comparison, Rudra and MirChecker can detect more types of bugs, but their capability in detecting memory deallocation issues is limited.

5.3.4 ***Usability.*** This subsection studies the usability of SafeDrop by examining whether it is useful to find new bugs from real-world Rust crates. To this end, we collect 50 vulnerable Rust crates from GitHub which employ unsafe constructors because unsafe constructors serve as a common component in most bug patterns presented in Listing 2. Therefore, we search projects from GitHub with the keyword "`from_raw`" or "`uninitialized`". "`from_raw`" can match many different constructors, such as `Box::from_raw()`, `String::from_raw_parts()`, and `Vec::from_raw_parts()`. We then employ SafeDrop to detect if these crates actually suffer from memory-safety issues. Note that we choose these crates based on arbitrary keywords because such programs are more vulnerable. However, our approach can be generalized to all Rust programs, *e.g.,* with other unsafe constructors.

Table 2 presents our experimental results after manually check. Overall, 12 out of 50 crates have memory-safety bugs, and there are only eight false positives generated by our tool. All these bugs can be verified with proof-of-concept cases. The result shows memory deallocation problems are very common among Rust crates using unsafe constructors. We have reported all bugs to developers on GitHub. 26 out of 103 bugs in three crates have been

```
#Real-world Bug Found by SafeDrop    // from crate: apres_bindings
pub fn get_ppqn( // interior unsafe function
    midi_ptr: *mut MIDI) -> u16 {
-   let mut midi = unsafe { Box::from_raw(midi_ptr) }; // unsafe constructor of Box<T>
+   let midi = mem::ManuallyDrop::new(Box::from_raw(midi_ptr));
+   // use smart pointer mem::ManauallyDrop<T> to avoid being dropped
    let output = midi.get_ppqn(); // double-free occurs if unwinding here
-   Box::into_raw(midi); // transfer to raw pointer to avoid being dropped
    output
}
```

(a) The bug sample in crate `apres_bindings`.

```
#Real-world Bug Found by SafeDrop    // from crate: stretch
pub unsafe extern "C" fn stretch_node_free( // unsafe FFI function
    stretch: *mut c_void, node: *mut c_void ) {
-   let mut stretch = Box::from_raw(stretch as *mut Stretch); // unsafe constructor of Box<T>
-   let node = Box::from_raw(node as *mut Node); // unsafe constructor of Box<T>
+   let stretch = mem::ManuallyDrop::new(Box::from_raw(stretch as *mut Stretch));
+   let node = mem::ManuallyDrop::new(Box::from_raw(node as *mut Node));
    stretch.remove(*node); // double-free occurs if unwinding here
-   Box::leak(stretch);
}
```

(b) The bug sample of FFI function in crate `stretch`.

Fig. 7. Example bugs found by SafeDrop.

confirmed by their developers. Figure 7a demonstrates one sample bug that contains a vulnerable program point between `Box::from_raw()` and `Box::into_raw()`. Unwinding at this point would cause double free. Similar bug patterns occur multiple times in another popular crate `stretch` (with 1.7k stars on GitHub). As shown in Figure 7b, `stretch` provides a function (`stretch_node_free()`) that can be called by other programming languages (Swift and Kotlin). As intended by the developer in the original code, the objects of `stretch` and `node` should not be owned/dropped by the Rust code. However, there are vulnerable points before the last statement is successfully executed. Panicking at these vulnerable points would cause double free. Although the function is labeled as unsafe, there could be better implementations (*e.g.*, using `ManuallyDrop`) without such vulnerabilities. Finally, we also apply Rudra, Miri, and MirChecker to these crates. Unfortunately, none of them can detect these memory deallocation bugs found by our tool. The result is consistent with those reported previously in Section 5.3.3.

5.4 Limitation and Discussion

Now we discuss the limitations of our approach.

5.4.1 ***False Positives.*** Our current implementation of inter-procedural analysis is based on the query `OptimizedMir` to obtain the MIR of the callees. However, some compiler internal trait implementations cannot be obtained by this query. For such cases, we establish alias relationships among the unfiltered arguments and the return value by default to be conservative. As an exception, we do not consider the alias set established from `Clone` trait, because the return value of the function `clone()` is a deep copy of the original variable and thus can be filtered [19]. Since these cases are rare and our type filter can discriminate most of the false aliasing issues, the remaining false positives could be manageable. This is the main reason that accounts for the false positives of our experimental results.

5.4.2 ***False Negatives.*** SafeDrop does not support pointer arithmetic, function pointer, or closure which may lead to some false negatives. Besides, our approach may suffer false negatives for some inline functions due to the implementation of the Rust compiler. Our query can obtain the MIR of such inlined functions but with empty information. All these issues may lead to losing some alias relationships.

6 RELATED WORK

This section compares our paper with related work, including existing work on Rust that can also detect memory-safety bugs and other approaches dedicated to memory-safety bug detection.

6.1 Existing Work on Rust

As Rust surges into popularity, it also attracts much attention from the research field, especially those on formal verification (*e.g.,* [6, 12, 18, 41]), usage of unsafe code [5], as well as how to deal with unsafe code [13, 23, 29]. Among all tools, Rudra [7] and MirChecker [28] are two static analysis tools that can also detect memory-safety bugs. We have already compared SafeDrop with them in Section 5.3.3. In general, SafeDrop is different from them in that it is a flow- and field-sensitive approach dedicated to detecting memory deallocation bugs. Our experimental results have already shown that SafeDrop is more powerful and accurate in detecting such bugs. Miri [31] is a dynamic analysis tool for Rust that can also detect memory-safety bugs. However, the effects of such dynamic analysis tools largely depend on the test cases. Besides, another tool MIRAI [30] is not directly applicable to our problem. MIRAI is originally invented as a tool for mitigating the weakness of Rust in implementing Diem. It mainly focuses on detecting possible program aborting with reachability analysis. It also supports taint analysis, and the main purpose is to detect information leakage bugs, which are different from ours.

6.2 Approaches for Other Languages

If not considering Rust, there are already dozens of existing papers and tools proposed to detect memory-safety issues. Such work can be classified into two folds. The first fold is tools proposed for other specific programming languages (*e.g.,* [11, 21, 22, 33, 44]. for C/C++) and is not directly applicable to Rust. The second fold is tools based on binaries or LLVM intermediate codes, such as Valgrind [32] and AddressSanitizer [36]. Both Valgrind and AddressSanitizer are dynamic binary analysis tools that suffer the similar soundness issue as Miri [31].

Among existing work, guarded value-flow or sparse value-flow [11, 39, 40] are the most closing ones that can also be used to solve the problem after being tailored for Rust. Guarded value-flow (a.k.a. FastCheck [11]) formulates the problem of memory leakage detection as to check if each memory allocation site can be paired with one reachable deallocation site in all cases. It can also be employed to detect double-free bugs by checking if one memory would be deallocated twice in one possible path. SABER [39, 40] is an improvement based on FastCheck. However, such approaches suffer critical limitations in two aspects and may not perform well for our problem. Firstly, these approaches require alias analysis for preprocessing. FastCheck directly employs an existing Steensgaard-style [38] method, and SABER employs Anderson-style [3]. Both these methods are inaccurate for our problem because they are flow-insensitive and path-insensitive. Because alias analysis is an NP-hard problem, the limitation cannot be easily tackled. On the contrary, our approach improves the alias analysis precision for Rust MIR to be both flow-sensitive and path-sensitive. The second critical limitation lies in how to accurately model the branch conditions, such as those challenges discussed for symbolic execution [43]. Our approach treats all paths as reachable except several fixed patterns discussed Section 4.1 and 4.5. It is more suitable for Rust because the conditions for triggering unwinding paths are very difficult to model by the existing approach. Besides, our work integrates several other subtle designs toward solving the problem well, such as Tarjan-based path extraction and fixed-point iteration for recursion. In contrast, SABER handles loops and recursions by simply bounding them to one iteration.

7 CONCLUSION

In this paper, we proposed a novel path-sensitive data-flow analysis approach to detect memory deallocation bugs for Rust programs. It has employed several subtle designs to be both precise and scalable, including a modified Tarjan algorithm for path-sensitive analysis and a cache-based strategy for inter-procedural analysis. We have

implemented and integrated the approach into the Rust compiler and conducted evaluation experiments with dozens of Rust crates. Results show our approach can effectively detect such bugs with only a small number of false positives and a very small overhead. Besides, our tool is more powerful in detecting such bugs than other existing tools. We believe the method proposed in our paper is not only useful for Rust but should also be able to inspire other programming languages with automatic memory deallocation mechanisms or RAII.

ACKNOWLEDGEMENT

This work was supported by Alibaba Group, CCF-ANT Fund (project id: RF20210017) and the Natural Science Foundation of Shanghai (No. 22ZR1407900).

REFERENCES

[1] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. 2007. *Compilers: principles, techniques, & tools.* Pearson Education India.
[2] Glenn Ammons and James R. Larus. 1998. Improving Data-Flow Analysis with Path Profiles. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98)*. Association for Computing Machinery, New York, NY, USA, 72–84.
[3] Lars Ole Andersen. 1994. *Program Analysis and Specialization for the C Programming Language.* Technical Report.
[4] Brian Anderson, Lars Bergstrom, Manish Goregaokar, Josh Matthews, Keegan McAllister, Jack Moffitt, and Simon Sapin. 2016. Engineering the servo web browser engine using Rust. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16)*. 81–89.
[5] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J Summers. 2020. How do programmers use unsafe rust? *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–27.
[6] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 30 pages.
[7] 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 (SOSP '21)*. 84–99.
[8] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. 2017. System Programming in Rust: Beyond Safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (Whistler, BC, Canada) *(HotOS '17)*. Association for Computing Machinery, New York, NY, USA, 156–161.
[9] Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: Probabilistic Memory Safety for Unsafe Languages *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 158–168.
[10] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. 2012. Undangle: Early Detection of Dangling Pointers in Use-after-Free and Double-Free Vulnerabilities *(ISSTA '12)*. Association for Computing Machinery, New York, NY, USA, 133–143.
[11] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. 2007. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. 480–491.
[12] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2019. RustBelt Meets Relaxed Memory. 4, POPL, Article 34 (2019), 29 pages.
[13] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust Typechecker Using CLP. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, Nebraska) *(ASE '15)*. IEEE Press, 482–493.
[14] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is Rust Used Safely by Software Developers?. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20)*. IEEE.
[15] 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 (STOC '83,)*. Association for Computing Machinery.
[16] David L Heine and Monica S Lam. 2003. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation (PLDI '03)*. 168–181.
[17] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked Borrows: An Aliasing Model for Rust. *Proc. ACM Program. Lang.* 4, POPL, Article 41 (Dec. 2019), 32 pages.
[18] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. 2, POPL, Article 66 (Dec. 2017), 34 pages.
[19] Steve Klabnik and Carol Nichols. 2019. *The Rust Programming Language (Covers Rust 2018).* No Starch Press.
[20] Stephen C Kleene and Emil L Post. 1954. The upper semi-lattice of degrees of recursive unsolvability. *Annals of mathematics* (1954), 379–407.

[21] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Model Checking for Weakly Consistent Libraries *(PLDI '19)*. Association for Computing Machinery, New York, NY, USA, 96–110.
[22] Erik Kouwe, Vinod Nigade, and Cristiano Giuffrida. 2017. DangSan: Scalable Use-after-free Detection.
[23] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. 2017. Sandcrust: Automatic Sandboxing of Unsafe Components in Rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems* (Shanghai, China) *(PLOS'17)*. Association for Computing Machinery, New York, NY, USA, 51–57.
[24] William Landi. 1992. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)* 1, 4 (1992), 323–337.
[25] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD conference*, Vol. 5.
[26] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. 2015. Preventing Use-after-free with Dangling Pointers Nullification. In *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS'15)*.
[27] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Shane Leonard, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. The Tock Embedded Operating System. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SenSys '17)*. 1–2.
[28] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. 2021. MirChecker: Detecting Bugs in Rust Programs via Static Analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*. 2183–2196.
[29] Peiming Liu, Gang Zhao, and Jeff Huang. 2020. Securing Unsafe Rust Programs with XRust. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 234–245.
[30] Mirai. 2021. Rust mid-level IR Abstract Interpreter. https://github.com/facebookexperimental/MIRAI
[31] Miri. 2019. An interpreter for Rust's mid-level intermediate representation. https://github.com/rust-lang/miri
[32] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.* 42, 6 (June 2007), 89–100.
[33] Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. Static detection of asymptotic performance bugs in collection traversals. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. 369–378.
[34] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding Memory and Thread Safety Practicesand Issues in Real-World Rust Programs. In *Proc. of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*.
[35] Tahina Ramananandro, Gabriel Dos Reis, and Xavier Leroy. 2012. A mechanized semantics for C++ object construction and destruction, with applications to resource management. *ACM SIGPLAN Notices* 47, 1 (2012), 521–532.
[36] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany
[37] Richard M Stallman. 2002. GNU compiler collection internals. *Free Software Foundation* (2002).
[38] Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '96)*. 32–41.
[39] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA '12)*. 254–264.
[40] Yulei Sui, Ding Ye, and Jingling Xue. 2014. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering (TSE)* 40, 2 (2014), 107–122.
[41] J. Toman, S. Pernsteiner, and E. Torlak. 2015. Crust: A Bounded Verifier for Rust (N). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*. 75–80.
[42] Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael Lyu. 2022. Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2022).
[43] Hui Xu, Zirui Zhao, Yangfan Zhou, and Michael R Lyu. 2018. Benchmarking the capability of symbolic execution tools with logic bombs. *IEEE Transactions on Dependable and Secure Computing* 17, 6 (2018), 1243–1256.
[44] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2018. Spatio-Temporal Context Reduction: A Pointer-Analysis-Based Static Approach for Detecting Use-after-Free Vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 327–337.