Detecting Memory Errors in Rust Programs Including Unsafe Foreign Code*

Andrea Franceschi[1], Letterio Galletta[1], and Pierpaolo Degano[1,2]

[1] IMT School Advanced Studies Lucca, Lucca, Italy
[2] University of Pisa, Pisa, Italy

Abstract

Memory corruption is one of the oldest and most disruptive problem in computer security, through which attackers may maliciously alter the program control flow. Unsafe languages, like C and C++, are prone to vulnerabilities of this kind. Recently, Rust has been proposed as a safe alternative, ensuring memory safety through proper compile-time checks with no penalties at run-time. However, the Rust compiler is not able to provide these guarantees when code written in an unsafe language is integrated into Rust code through a *Foreign Function Interface* mechanism. In this way, the memory corruption vulnerabilities that Rust aims to eliminate may be re-introduced. Here, we propose a static taint analysis that targets both Rust and foreign code to detect the common memory errors *use-after-free*, *never-free*, and *double-free*. We define our analysis (abstract domain and transfer functions) and algorithms for computing the taint propagation and detecting possible memory errors.

Keywords: Bug Detection, Unsafe Rust, Taint Analysis.

1 Introduction

Memory corruption vulnerabilities are one of the most old and disruptive problems in computer security. They have been often caused by unsafe languages such as C and C++, which typically expose raw pointers and allow programmers to manage the memory manually, making them prone to memory errors. The lack of memory safety in such languages enables attackers to exploit memory bugs by maliciously altering the program behavior or taking full control over the control flow.

Rust is an emerging programming language developed to ensure memory safety using a strong type system and proper compile-time checks with no penalties at runtime. To achieve that, Rust implements an automatic ownership-based memory management mechanism that requires no garbage collector, making this language suitable for system-level applications [20]. The promise of providing safety with no cost at runtime has led many companies and open source communities to re-write their software in Rust, among which Firefox [11] and Linux kernel [17]. However, this porting is happening gradually, making such software multi-language, in which developers integrate the existent code with Rust.

Rust offers the mechanisms of `unsafe` blocks and of *Foreign Function Interface* (FFI) to allow the integration of different languages. The first mechanism temporally suspends the safety checks performed by the Rust compiler on the blocks declared unsafe, while the second one allows the linking of foreign code to a Rust program. Since the foreign code is likely to be written in memory unsafe languages, its inclusion may re-introduce memory corruption vulnerabilities, making FFI the most common cause of memory issues in Rust [28].

Consider as an example the code of Figure 1 where we allocate a memory cell in the heap through a smart pointer (line 5). After converting it to a raw pointer `z_raw` and forgetting its ownership (line 7), we pass `z_raw` to a C function through an FFI call. The C function casts its argument into an integer

*Oral communication, not to be included in the conference proceedings.

```rust
use std::ffi::c_void;
extern "C" {fn cast_and_free_pointer(ptr: *mut c_void);}
fn main() {
    // allocate memory on the heap and store it in a Box
    let z = Box::new(90);   // z -> ALLOC
    // convert Box to a raw pointer, forgetting the ownership
    let z_raw: *mut c_void = Box::into_raw(z) as *mut c_void; // z -> MV
    unsafe {
        // pass the raw pointer to an FFI function
        cast_and_free_pointer(z_raw);
        // z_raw is pointing to freed memory
    }
    // use the pointer after it has been freed
    unsafe {
        if !z_raw.is_null() {
            let int_ptr = z_raw as *mut i32;
            println!("Value after free: {}", *int_ptr);
        }}}
```

Rust code that passes a raw pointer to a C FFI and uses it after being freed from that function.

```c
#include <stdio.h>
#include <stdlib.h>
void cast_and_free_pointer(void *ptr) {
    if (ptr != NULL) {
        // cast the void pointer to an integer pointer
        int *int_ptr = (int *)ptr;
        printf("Value: %d\n", *int_ptr);
        // free the allocated memory
        free(int_ptr);
    }}
```

C function printing the content pointed to by a raw pointer and freeing the memory.

Figure 1: Rust program interacting with a C FFI demonstrating a use-after-free scenario.

pointer and frees the corresponding memory. When the Rust code accesses the value pointed by `z_raw` (line 17), the operation produces a *use-after-free* error since the C function has deallocated the memory.

Since the Rust borrow checker is turned off within unsafe blocks, and since the Rust compiler only links the included code, the burden of proving a Rust program using FFI safe is left to the programmer. It is thus important to support programmers in mechanically detecting potential vulnerabilities when Rust programs include foreign code. To this aim, we propose here a sound static analysis, namely a taint analysis, that targets both Rust and foreign code.

Our taint analysis can be easily tailored to various different cases, but here we restrict to C as external language and to the common memory errors *use-after-free*, *never-free*, and *double-free*. More precisely, our analysis works as follows. First, we construct an inter-procedural Control Flow Graph (ICFG) capturing the interactions between Rust and C functions. To build such ICFG, we exploit the information that Rust and C compilers make us available: the Mid-level Intermediate Representation for Rust [14], and the Intermediate Representation of LLVM for C [1]. We define an abstract domain to track taint information related to heap management, focusing specifically on memory objects passed through FFI and their ownership. An abstract state is then defined by associating an element of this domain with each variable and basic block. As is customary, the effect of executing a statement is modeled abstractly in the analysis using a transfer function [7]. This function updates the taint information in the abstract state by accounting for changes in heap allocations and ownership transfers, as dictated by the semantics

of the program statements. We then use a fixed-point algorithm to propagate the taint information across the ICFG nodes iteratively. By analyzing the taint information resulting from this propagation, we can detect the presence of possible memory management issues.

Back to our example, the analysis first assigns the abstract value *ALLOC* to the variable `z`, indicating that a cell in the heap has been allocated. Due to the statement `Box::into_raw(z)`, the abstract state is updated to associate `z` with the abstract value *MV*, denoting that its ownership has been forgotten. The analysis detects then a path in the ICFG where the control reaches a call to the `free` function on an allocated heap cell, causing its deallocation. Moreover, the analysis also discovers that the same memory cell is later used again in another Rust statement. As a result, a *use-after-free* error is detected.

Summing up, the main contributions of this paper are:

- We define a generic analysis schema to detect memory errors in Rust programs using FFI;
- We instantiate the above schema to detect use-after-free, never-free, and double-free memory errors when C is used as a foreign language;
- We define algorithms for program analysis and vulnerability detection.

In the rest of the paper, we proceed as follows. In Section 2, we briefly recall the basics of Rust, its compiler, and its intermediate representations. The same section also provides an overview of abstract interpretation and taint analysis. Section 3 describes the algorithm to build the ICFG, defines our abstract domain with the needed transfer functions, and presents the analysis and the detection algorithms. In Section 4, we compare our proposal with the relevant related work, while in Section 5, we draw some conclusions and overview future work.

2 Background

Rust Rust's most distinctive feature is the so-called *ownership* model that enables safe memory management without a garbage collector. It consists of a set of rules that govern how a program manages memory [16], ensuring memory allocations and deallocations are handled at compile time. In Rust, each value has a unique *owner* that governs its lifetime: when the owner goes out of scope, the owned value is dropped. To allow sharing values between structures, functions, and threads, Rust offers the *borrowing* mechanism that allows creating *immutable* or *mutable* references to values (*borrows*). Immutable references allow aliasing, but all memory locations reachable through them remain unchanged along with the borrow. These references can be copied, possibly originating new immutable borrows. Instead, mutable references allow changing the referenced memory location. In this case, Rust enforces exclusivity: no other references (mutable or immutable) can access the memory while the mutable borrow is active. Rust enforces the ownership rules through the mechanism of *lifetimes* that are named regions of code in which a reference is valid during the execution of the program [2], therefore preventing *dangling references* [16]. The entity responsible for enforcing statically the above rules is the *borrow checker*. Technically, it compares scopes to determine whether all borrows are valid, and it operates on the Mid-level Intermediate representation (MIR), an internal intermediate representation of the Rust compiler `rustc`. However, the borrow checker suspends its checks within `unsafe` blocks, assuming that they are fine, and ignores foreign code.

Table 1 shows the syntax of a core Rust MIR considered in our work. A basic block *bb* consists of a possibly empty sequence of statements followed by a terminator ending the block and changing the program flow. The binary operations $\oplus$ are the usual ones, and their operands o are constant values $z \in Z$ or places $x@p$. The set *Lifetime* of lifetimes l is partially ordered by inclusion of the code regions they denote. A local v corresponds to a stack location i.e., function arguments, temporaries and local

Constant	z	$\in$	Z
BasicBlock	bb	::=	$s.t \mid t$
Operation	$\oplus$	::=	$+ \mid \times \mid \ldots$
Operand	o	::=	$z \mid x@p$
Lifetime	l	$\in$	$\mathscr{L}$
Local	v	$\in$	$\{v_0, \ldots, v_n\}$
Qualifier	q	::=	$\texttt{imm} \mid \texttt{mut}$
Place	$x@p$	::=	$v \mid x@p[v] \mid *x@p \mid x@p.field$
Rvalue	e	::=	$\& \; l \; q \; x@p \mid o \mid o_1 \oplus o_2 \mid \ldots$
Statement	s	::=	$s_1.s_2 \mid \texttt{nop} \mid x@p \leftarrow e \mid x@p \leftarrow \texttt{new} \mid \ldots$
Terminator	t	::=	$\texttt{call(F, } o_1, \ldots, o_n\texttt{)} \mid \texttt{drop}(x@p) \mid \ldots$

Table 1: The syntax of a core of Rust MIR.

variables. The distinguished local v_0 stores the return value of a function. A qualifier q indicates whether a variable is mutable or immutable. A place $x@p$ refers to a memory location of a variable x and a path p to reach it, and provides the means to access or modify the content of the variable x. A place may either be a local v or a projection, i.e., an operation to project out from a base place, typically used to refer to an array element $x@p[v]$ or to a field ($x@p.field$), or to dereference a pointer ($*x@p$). Rvalues produce a value, and they can be plain expressions $o_1 \oplus o_2$ or references to mutable or immutable places $\& \; l \; q \; x@p$. The statements we consider are standard and include the no-operation `nop`; sequencing $s_1.s_2$; assignment; heap allocation $x@p = \texttt{new}$ that initializes a fresh pointer to the new object. the terminators include the default function $\texttt{drop}(x@p)$ that explicitly deallocates the memory of $x@p$, function calls $(F, o_1, \ldots, o_n)$, which invokes the function F and stores the return value in the local v_0, and jumps and branches that we omit here for brevity.

Abstract Interpretation Abstract interpretation is a theory for designing approximate semantics of programs that can be used to provide sound answers to questions about their run-time behavior [8]. Abstract interpretation is grounded in lattice theory and Galois connections [23]. A lattice $(L, \sqsubseteq, \sqcup, \sqcap)$ is a partially ordered set $(L, \sqsubseteq)$ such that $\forall v, w \in L$ there exists a least upper bound $v \sqcup w$ and a greatest lower bound $v \sqcap w$. Let $(L_1, \sqsubseteq_2)$ and $(L_2, \sqsubseteq_2)$ be posets, (α, γ) is a Galois connection between L_1 and L_2, in symbols $(L_1, \sqsubseteq_1) \underset{\alpha}{\overset{\gamma}{\rightleftharpoons}} (L_2 \sqsubseteq_2)$, if and only if $\alpha \in L_1 \to L_2, \gamma \in L_2 \to L_1$ and $\forall v \in L_1, \forall w \in L_2, \alpha(v) \sqsubseteq_2 w \iff v \sqsubseteq_1 \gamma(w)$. The two properties $\alpha(v) \sqsubseteq_2 w$ and $v \sqsubseteq_1 \gamma(w)$ both mean that w is a *correct approximation* of the concrete element v and we say what: $\alpha(v)$ is the most precise approximation of $v \in L_1$ in L_2 and $\gamma(w)$ is the least precise element of L_1 which can be correctly approximated by $w \in L_2$. Abstraction allows static analysis to work with manageable approximations of complex properties of a program, while concretization ensures that these approximations can be translated back into meaningful, concrete terms. In this approach, program states are represented in an *abstract domain* [21], where each abstract element represents a set of possible concrete states. Abstract operations [22], called *transfer functions*, are defined on the abstract domain to describe how program statements transform states. The result of an analysis is typically defined as the *fixed point* of a monotonic *transfer function* [5] $f \colon L \to L$, where L is a complete lattice [12]. When L satisfies the *ascending chain condition* [15], i.e., every ascending sequence of elements of L eventually stabilizes, the *least fixed point* of f is computed through an iterative process $(f^n(\bot))_n$ until further iterations do not yield any new information. When L does not satisfy the ascending chain condition, an approximation of the least fixed point can be anyway computed by resorting to a widening operator [6].

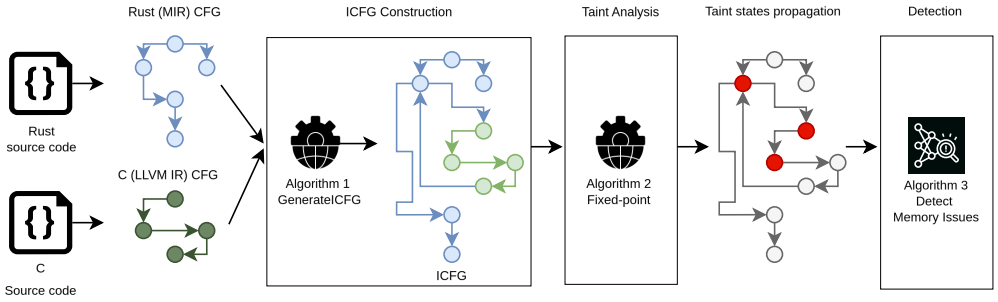

Figure 2: An overview of our cross-language static taint analysis.

Taint analysis Taint analysis is a common form of information-flow analysis [26] that captures mainly the explicit flows, and that can be both static and dynamic [3]. The analysis reasons on the control and data dependence from a *source statement* to a *sink statement* [27]. The *source-sink* analysis allows information flow checking to detect software bugs via value-flow reachability [25]. The analysis is formalized as follows: let *Var* be the set of all variables of a program and S be the set of all the statements. We define a source as a pair, written $v_{src}@s_{src}$, where s_{src} is the statement that introduces the variable v_{src}. Similarly, a sink is pair $v_{snk}@s_{snk}$ where v_{snk} is a variable and s_{snk} is the statement that *uses* it. Traveling along the control flow, one checks the reachable sinks from a given source and analyses the (abstract) effects caused by the current (abstract) execution.

3 Analysis Design

Figure 2 shows the pipeline stages we use for computing our taint analysis and for detecting possible memory errors. First, we exploit the Rust compiler for extracting the MIR from the Rust code and the `clang` compiler for extracting the LLVM IR from the C code. Then, we compute the CFGs from these IR representations and combine them into a single ICFG by our Algorithm 1. We analyze the resulting ICFG through Algorithm 2 that iteratively propagates the taint information to each node. Finally, we inspect the obtained ICFG annotated with the analysis results by Algorithm 3 to detect variables that are either freed multiple times, used after being freed, or never freed at all. We detail each step below.

Inteprocedural Control-flow Graph Construction The starting point of our analysis is the interprocedural CFG that captures the transfer of control between the basic blocks of Rust and C functions. Algorithm 1 combines the CFGs of the Rust and the C code generated by the corresponding compilers as follows. For each FFI call in the Rust CFG, the algorithm first links the corresponding entry point of the C CFG, and then ensures that the execution flow returns correctly to the Rust CFG after the C function has completed its execution. Finally, the algorithm returns the ICFG, which represents the combined control flow of both the Rust and C components. Note that since the Rust compiler has no information about the FFI calls, Algorithm 1 parses the Rust source file to identify all functions declared in the `extern` block containing the FFI targets.

Abstract domain for memory management Our analysis over-approximates the behavior of a program by running it on an abstract domain, each element of which represents an *abstract state* of the execution. This abstract state keeps track of the ownership of the memory cell in the heap assigned to

Algorithm 1 GenerateICFG

```
1:  function GENERATEICFG(rust_code, c_code)
2:      ffi_functions = PARSERUSTSOURCEFORFFI(rust_code)
3:      rust_mir = COMPILETOMIR(rust_code)
4:      (rust_cfg, ffi_calls) = PARSERUSTMIRTOCFG(rust_mir, ffi_functions)
5:      c_ir = COMPILETOLLVMIR(c_code)
6:      (c_cfg, c_entry_points) = PARSELLVMIRTOCFG(c_ir)
7:      icfg = rust_cfg
8:      for each ffi_call in ffi_calls do
9:          c_entry = FINDENTRYPOINT(c_cfg, ffi_call)
10:         c_exit = FINDEXITPOINT(c_cfg, c_entry)
11:         ADDEDGE(icfg, ffi_call, c_entry)
12:         ADDEDGE(icfg, c_exit)
13:         NEXTNODE(rust_cfg, ffi_call)
14:     end for
15:     return icfg
16: end function
```

each variable in each basic block — recall that heap references passed to a foreign function are either related by mutable or immutable borrows, or otherwise the ownership is forgotten.

Before defining our abstract states, we introduce an abstract representation for the value of a memory cell in the heap, giving rise to a value abstract domain. To achieve that, we define the lattice *Cell_value*

$$\mathit{Cell\_value} = (\{\bot, \mathit{ALLOC}, \mathit{FREED}, \mathit{MB}, \mathit{IMMB}, \mathit{MV}, \top\}, \sqsubseteq)$$

ordered according to the Hasse diagram of Figure 3 [13]. The bottom element $\bot$ (the minimum element of the lattice) represents the undefined value; the allocation abstract value *ALLOC* represents the case when a variable is assigned to a heap memory cell, while *FREED* stands for a cell allocated to a variable and then freed; the borrowed abstract value *MB* (*IMMB*, resp.) represents the case when the heap memory cell is passed through an FFI function as a mutable (immutable, resp.) reference; the moved abstract value *MV* records that a heap memory cell is passed to a foreign function, forgetting its ownership (recall that `rustc` is not responsible for errors and side effects involving that variable). Finally, the element $\top$ (the maximum element of the lattice) represents the case when we do not have precise information on the status of the heap cell.

An abstract memory is a mapping $\sigma\colon \mathit{Var} \to \mathit{Cell\_value}$ that associates each variable v with a cell value. Let the set of abstract memories *A_mem* be pointwise ordered according to the partial ordering relation $\sqsubseteq$ (overloading the notation) defined as $\forall \sigma_1, \sigma_2 \in \mathit{A\_mem}, \sigma_1 \sqsubseteq \sigma_2 \iff \forall v \in \mathit{Var}, \sigma_1(v) \sqsubseteq \sigma_1(v)$. Similarly, the join operator $\sqcup$ combines two abstract states $\sigma_1, \sigma_2 \in \mathit{A\_mem}$ in a pointwise manner: $\forall v \in \mathit{Var}\ (\sigma_1 \sqcup \sigma_2)(v) = \sigma_1(v) \sqcup \sigma_2(v)$. By returning the least upper bound of the memory values associated with each variable v, the join operator combines the information of both σ_1 and σ_2, and represents the most precise state that conservatively includes both.

Finally, we define an abstract state $as\colon B \to \mathit{A\_mem}$ as a mapping from basic blocks $b \in B$ to abstract memories describing the memory cell in the heap assumed by the variables in b. Once more, the lattice of abstract states $(\mathit{Abs\_state}, \sqsubseteq)$ is pointwise partially ordered.

Transfer functions The transfer functions update the abstract states of a program according to the semantics of the statement under execution [7]. To define our transfer functions, we proceed as follows. First, we display in Table 2 (left part) the abstract semantics of expressions, given an abstract memory

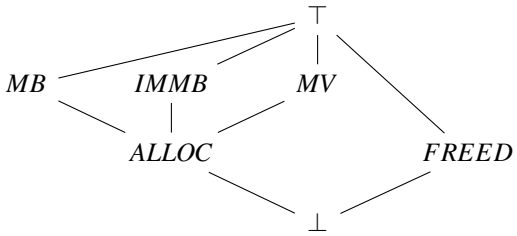

Figure 3: *Cell_value* partial order.

$$
\begin{array}{ll}
[\![e]\!]^a_\sigma = \sigma(e) & S[\![nop]\!]^a_\sigma = \sigma \\
[\![\&\ l\ \text{imm}\ x@p]\!]^a_\sigma = \mathit{IMMB} & S[\![s_1.s_2]\!]^a_\sigma = S[\![s_2]\!]^a_{S[\![s_1]\!]^a_\sigma} \\
[\![\&\ l\ \text{mut}\ x@p]\!]^a_\sigma = \mathit{MB} & S[\![x@p \leftarrow \text{new}]\!]^a_\sigma = \sigma\,[x@p \mapsto \mathit{ALLOC}] \\
[\![e]\!]^a_\sigma = \bot & S[\![x@p \leftarrow e]\!]^a_\sigma = \sigma\,[x@p \mapsto [\![e]\!]^a_\sigma] \\
 & S[\![\text{drop}\ x@p]\!]^a_\sigma = \sigma\,[x@p \mapsto \mathit{FREED}] \\
 & S[\![\text{call}(F, o_1, \ldots, o_n)]\!]^a_\sigma = \sigma'\,[v_0 \mapsto v] \\
 & \qquad \text{where } (v, \sigma') = S'[\![B_F]\!]^a_{\sigma[x_1 \mapsto [\![o_1]\!]^a_\sigma, \ldots, x_n \mapsto [\![o_n]\!]^a_\sigma]}
\end{array}
$$

Table 2: Abstract semantics for Rvalues (left) and transfer functions for statements (right).

σ. If the expression is a place, it returns the corresponding abstract value in the memory; if the expression is an immutable (resp. mutable) reference, the function returns the corresponding abstract value *IMMB* (resp. *IMMB*). Otherwise, it returns the bottom element to denote that its value does not concern references in the heap.

Then, we define the transfer functions for the statements that are relevant to our analysis in Table 2 (right part). We briefly comment on their definition below. A `nop` statement does not operate on the heap, so its transfer function is simply the identity. As expected, the transfer function for the sequential composition $s1.s2$ composes the results of its components. The transfer function for the `new` statement for allocating heap memory assigns the place $x@p$ the abstract state *ALLOC*. An *assignment* $x@p \leftarrow e$ causes the entry for the place $x@p$ in the current abstract memory to be updated to the abstract value of the evaluated rvalue e. The execution of the primitive function `drop` deallocates the memory cell assigned to $x@p$, so its entry in the abstract memory σ is assigned the abstract value *FREED*. The transfer function for a statement calling the function F requires a few steps, in a call-by-value fashion. First, the actual parameters o_i are abstractly evaluated in the current abstract state σ, and the formal parameters x_i are assigned the computed abstract values $[\![o_i]\!]^a_\sigma$. Then, the body B_F of F is evaluated with the transfer function $S'[\![\_]\!]$ (similar to $S[\![\_]\!]$ but that additionally produces the returned value), resulting in a value v and in a (possibly) new abstract state σ'_b. The final result is this new abstract memory updated by assigning the abstract value v to the reserved local v_0.

Propagating and checking taint information We use a taint analysis to detect whether a memory cell is accessed after being freed, freed multiple times, or never freed at all. Our proposal leverages a worklist algorithm to propagate taint information across the ICFG iteratively. Intuitively, Algorithm 2 works as follows. Each variable in a basic block is associated with an abstract state and a multi-set of taint values. The taint value is one of the following: `assign`, `free`, `use`, and indicates how the variable

is affected by the instructions in the basic block.

Initially, we set the abstract state and the taint value to $\bot$ and $\emptyset$, respectively, to indicate the default state and no taint. Since the worklist will contain the nodes of the ICFG to be processed, we also initialize it with the ICFG entry node, the dummy node `main` standing for the main function. Until the worklist is not empty, the algorithm dequeues the node and processes it by computing the corresponding transfer function. For each successor node, the algorithm checks if the new abstract state differs from the current state in which case it gets updated, as well as the taint state according to the new abstract state. Finally, it enqueues the current successor node in the worklist for further processing. The loop continues until the worklist is empty, meaning that the fixed point has been reached because no more updates are being propagated through the ICFG.

The memory bug detection phase associates a set of sources and sinks to each node of the ICFG. In this way, given a tainted source $v_{src}@s_{src}$ in a node of the ICGF, a sink $v_{snk}@s_{snk}$ in another node is also tainted if there is a path in the ICFG from the node that contains the statement s_{src} to the node that contains the statement s_{snk}. This sort of reachability analysis is the basis of our detection in Algorithm 3 that works as follows. The `detect_mem_issue` function inspects the entries of the `TaintStates` dictionary, where each entry corresponds to a program basic block, and classifies them into different types of memory-related issues. The algorithm iterates over each path p starting from a basic block b, and for each variable v in b, examines its taint state. If v occurs in a successor of b in p and is associated in both with `free`, we detect a double-free error because it has been deallocated more than once. Similarly, if v is associated with a non-empty multi-set with no `free`, we discover that the variable has been allocated but never freed. Finally, a user-after-free error is detected when v is freed in b and used in a successor of b in p. The returned multisets identify the variables that may potentially cause the corresponding memory errors.

4 Related Work

In the literature, several studies use MIR to detect potential vulnerabilities in Rust code. For instance, MirChecker [18] combines numerical static analysis with symbolic execution to identify runtime panics and lifetime corruption issues. Another proposal is SafeDrop [9] that leverages a modified Tarjan algorithm for path-sensitive analysis and uses a cache-based strategy for inter-procedural analysis to identify memory deallocation bugs in pure Rust programs, excluding FFI, that are caused by unsafe code. RCanary [10], instead, defines a model checker to detect leaks across the semi-automated boundary within Rust programs, implemented as an external component in Cargo.

MirChecker [18] and FFIChecker [19] are the proposals that are more similar to our work, as they both rely on an abstract domain, transfer functions, and a fixed-point algorithm, though applied in different ways. The abstract domain of MirCheker contains both numerical and symbolic values, and before executing a fixed-point algorithm, it generates a weak topological ordering (WTO) of the basic blocks within the CFG for analyzing unexpected program panic during normal execution. Instead, FFIChecker focuses on memory issues across the Rust/C FFI mechanism. Its analysis starts at the LLVM IR level of both Rust and C codes. However, its abstract domain does not track whether a borrow is mutable or immutable. In contrast, our model considers this distinction between providing more accurate tracking of heap-memory objects' abstract states, improving thus the analysis results. By capturing more detailed insights, it becomes easier to identify the root cause of an error and facilitate quicker fixes. In addition, we provide a formal definition of transfer functions, particularly useful for handling function calls. By introducing a context that contains information about each function's signature and behavior, we can map functions to their corresponding types and behaviors. This modular approach allows us to incrementally adapt the analysis when new functions are introduced in the standard library. Classifying function behavior also enhances the precision of the analysis, making it more adaptable and effective.

Algorithm 2 Fixed-point algorithm(ICFG)

AbstState ← Dict() ▷ Contains $\sigma_b \in A\_mem$ for each block b
TaintStates ← Dict() ▷ Contains the taint of each variable in each block
Worklist ← Queue(`main`) ▷ Initialize the Worklist with the main block
for each basic block b in ICFG **do**
  for each variable v in b **do**
    $AbstState[b][v] \leftarrow \bot$ ▷ Initialize abstract states
    $TaintStates[b][v] \leftarrow \emptyset$ ▷ Initialize taint states as empty set
  end for
end for
Worklist.enqueue(*entry_node*)
while *Worklist* is not empty **do**
  curr_node ← *Worklist*.dequeue()
  for each *succ_node* in *curr_node.successors* **do**
    for each variable v in *succ_node* **do**
      new_abst_state ← $S[\![curr_node]\!]^a$AbstState ▷ where s is the code of *succ_node*
      if new_abst_state ≠ AbstState[*Current_node*][v] **then**
        $AbstrStates[succ\_node][v]$ ← new_abst_state
      end if
      new_taint_st ← UPDATE_TAINT_STATE(v, *curr_node*, *TaintStates*, *AbstState*)
      if *nw_taint_st* ≠ $TaintStates[Current\_node][v]$ **then**
        $TaintStates[succ\_node][v] \leftarrow new\_taint\_st$
      end if
    end for
    Worklist.enqueue(*succ_node*)
  end for
end while
return DETECT_MEM_ISSUES(ICFG, *TaintStates*)

Algorithm 3 Detect memory issues

function DETECT_MEM_ISSUES(ICFG, *TaintStates*)
  use_after_free ← MultiSet()
  $double\_free$ ← MultiSet()
  $never\_free$ ← MultiSet()
  for each path p starting from block b in icfg **do**
    for each variable v in $TaintStates[b]$ **do**
      if $\exists b' \neq b$ in p s.t. $TaintStates[b][v] \cap TaintStates[b'][v] \supseteq \{\texttt{free}\}$ **then**
        $double\_free$.add(v)
      else if $TaintStates[b][v] \neq \emptyset \wedge \forall b' \neq b$ in p `free` $\notin TaintStates[b'][v]$ **then**
        $never\_free$.add(v)
      else if $TaintStates[b][v] \neq \emptyset \wedge \exists b' \neq b$ in p `use` $\in TaintStates[b'][v]$ **then**
        use_after_free.add(v)
      end if
    end for
  end for
  return $use\_after\_free, double\_free, never\_free$
end function

5 Conclusion

Here, we proposed a static taint analysis that targets Rust programs using foreign code through the FFI mechanism. We tailored our analysis to consider C as an external language and detect the common memory errors use-after-free, never-free, and double-free. To run the analysis, we built an ICFG to capture the interactions between Rust and C functions, exploiting the information that Rust and C compilers make us available. Our analysis relies on an abstract domain designed to track taint information related to heap management, focusing specifically on memory objects passed through FFI and their ownership. Moreover, we defined transfer functions to update such taint information in the abstract state as dictated by the semantics of the program statements. Finally, we presented a fixed-point algorithm that iteratively propagates the taint information across the ICFG nodes, and an algorithm that uses the taint information to detect the presence of possible memory management issues.

There are several future research directions. First, to make the analysis more precise, we can extend the abstract domain to consider explicitly annotated lifetimes, which would affect the transfer function definition. Concerning the C side, before the creation of the ICFG, we could improve the precision of the analysis by computing points-to information directly on the LLVM IR. Including a module responsible for pointer analysis would improve precision and impact the solution's scalability, on the other hand. Andresen's pointer analysis [4] offers a precise, context-insensitive inter-procedural analysis that is, in the worst-case scenario, cubic in the size of the number of nodes generated by its constraints. In contrast, Steensgaard's approach [24] offers a less precise but more time-efficient analysis, which is almost linear in the number of the program's statements. The effective implementation of this framework will be followed in the future, showing that our results will have concrete applications in the context of program analysis. Additionally, an experimental comparison with existing solutions will be intriguing.

References

[1] The LLVM compiler infrastructure. *https://llvm.org/*.

[2] The Rustonomicon. *https://doc.rust-lang.org/nomicon/*.

[3] Ashish Aggarwal and Pankaj Jalote. Integrating static and dynamic analysis for detecting vulnerabilities. In *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, volume 1, pages 343–350. IEEE, 2006.

[4] Lars Ole Andersen. Program analysis and specialization for the c programming language. 1994.

[5] Viviane Baladi and Gerhard Keller. Zeta functions and transfer operators for piecewise monotone transformations. *Communications in mathematical physics*, 127:459–477, 1990.

[6] Agostino Cortesi. Widening operators for abstract interpretation. In *2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pages 31–40. IEEE, 2008.

[7] Patrick Cousot. *Principles of abstract interpretation*. MIT Press, 2021.

[8] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of logic and computation*, 2(4):511–547, 1992.

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

[10] Mohan Cui, Hui Xu, Hongliang Tian, and Yangfan Zhou. rCanary: Detecting memory leaks across semi-automated memory management boundary in rust. *IEEE Transactions on Software Engineering*, 2024.

[11] Bushev D. How much Rust in Firefox? *https://4e6.github.io/firefox-lang-stats/*, 2024.

[12] Anne C Davis. A characterization of complete lattices. *Pacific J. Math*, 5(2):311–319, 1955.

[13] Peter Eklund, Jon Ducrou, and Peter Brawn. Concept lattices for information visualization: Can novices read line-diagrams? In *International Conference on Formal Concept Analysis*, pages 57–73. Springer, 2004.

[14] Rust Compiler Development Guide. The MIR (Mid-level IR). *https://rustc-dev-guide.rust-lang.org/mir/index.html*, 39.

[15] Michiel Hazewinkel, Nadiya Gubareni, and Vladimir V Kirichenko. *Algebras, Rings and Modules: Volume 1*, volume 575. Springer Science & Business Media, 2006.

[16] Steve Klabnik and Carol Nichols. *The Rust programming language*. No Starch Press, 2023.

[17] Hongyu Li, Liwei Guo, Yexuan Yang, Shangguang Wang, and Mengwei Xu. An empirical study of {Rust-for-Linux}: The success, dissatisfaction, and compromise. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 425–443, 2024.

[18] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. MirChecker: detecting bugs in Rust programs via static analysis. *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*, pages 2183–2196, 2021.

[19] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. Detecting cross-language memory management issues in Rust. In *European Symposium on Research in Computer Security*, pages 680–700. Springer, 2022.

[20] Nicholas D Matsakis and Felix S Klock. The Rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, 2014.

[21] Flemming Nielson and N Jones. Abstract interpretation: a semantics-based tool for program analysis. *Handbook of logic in computer science*, 4:527–636, 1994.

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

[23] Zahava Shmuely. The structure of Galois connections. *Pacific Journal of Mathematics*, 54(2):209–225, 1974.

[24] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, 1996.

[25] Yulei Sui and Jingling Xue. SVF: interprocedural static value-flow analysis in LLVM. pages 265–266, 2016.

[26] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. *ACM Sigplan Notices*, 44(6):87–97, 2009.

[27] Teng Wang, Haochen He, Xiaodong Liu, Shanshan Li, Zhouyang Jia, Yu Jiang, Qing Liao, and Wang Li. ConfTainter: Static taint analysis for configuration options. pages 1640–1651, 2023.

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