Detecting memory errors in rust programs including unsafe foreign code

Detecting Memory Errors in Rust Programs Including Unsafe Foreign Code

Andrea Franceschi[1 [0009−0002−6691−6145]], Letterio Galletta[1 [0000−0003−0351−9169]], and Pierpaolo Degano[1,2 [0000−0002−8070−4838]]

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

Abstract. Memory corruption is one of the oldest and most disruptive problems in computer security, through which attackers may maliciously alter the program control flow. Unsafe languages, such as C and C++, are prone to these types of vulnerability. A promising alternative is Rust, which ensures 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 programmers use Rust unsafe features or integrate code written in an unsafe language through the *Foreign Function Interface* mechanism. If the unsafe features and the integration of unsafe code are not handled with extreme care, the memory errors that Rust aims to eliminate may be reintroduced. Here, we define 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*, and implement it in the tool CREMA. Our experimental evaluation on real cases from GitHub shows that CREMA detects memory errors effectively.

Keywords: Bug Detection · Unsafe Rust · Static Taint Analysis.

1 Introduction

Memory corruption vulnerabilities are one of the oldest and most disruptive problems in computer security. They often occur in programs written in unsafe languages such as C and C++, which typically expose raw pointers and allow programmers to manage the memory manually, making it easy to introduce memory errors. Exploiting these errors enables attackers to maliciously alter the program's behavior or take full control over the program's 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 with no garbage collector, making it suitable for system-level applications [29]. The promise of providing safety with no cost at runtime has led many companies and open source communities to rewrite their software in Rust, among which Firefox [16] and the Linux kernel [25]. However, this porting is happening gradually: developers are integrating the existing

code with Rust, making their codebase multi-language. However, if this gradual porting is not done with extreme care, it can be detrimental to security [30].

Rust offers the mechanisms of `unsafe` blocks and of *Foreign Function Interface* (FFI) for the integration of different languages. The first mechanism temporarily suspends the safety checks performed by the Rust compiler and allows memory manipulation through raw pointers. A careless use of the unsafe features may lead to subtle memory issues. While FFI allows linking foreign code to a Rust program. Since the foreign code is likely to be written in a memory unsafe language, its inclusion may reintroduce memory corruption vulnerabilities, making FFI the most common cause of memory issues in Rust [51].

When programmers use one of the above mechanisms, they also have the burden to verify that their code is still safe. It is therefore important to support them in mechanically detecting potential vulnerabilities in these situations. To this aim, we propose 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 ourselves to C as the external language and to the common memory errors *use-after-free*, *never-free*, and *double-free*.

More precisely, to define our analysis we proceed as follows. First, we construct an *inter-procedural Control Flow Graph* (ICFG) capturing the interactions between Rust and C functions. To build such a graph, we exploit the information that Rust and C compilers make us available: the *Mid-level Intermediate Representation* for Rust [39], and the intermediate representation of LLVM for C [1]. Then, 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 defined by associating an element of this domain with each variable and basic block. As usual, the analysis models abstractly the effect of executing a statement using a transfer function [50]. 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 language. We then use a fixed-point algorithm to iteratively propagate the taint information across the ICFG nodes. By analyzing the resulting taint information, we detect possible memory issues.

We implement our analysis in the tool CREMA and apply it to scrutinize several Rust projects using FFI from GitHub, as well as in pure Rust code when unsafe blocks are present. Our experiments show that the tool is effective in detecting memory errors: CREMA detected 17 new memory leaks in some of these projects.

Summing up, the main contributions of this paper are:

- a generic analysis schema to detect memory errors in Rust programs using FFI and unsafe blocks;
- the application of the above schema to detect use-after-free, never-free, and double-free memory errors when C is used as a foreign language;
- the algorithms for program analysis and vulnerability detection;
- the analysis tool CREMA and the assessment of its effectiveness on 20 Rust projects from GitHub, where it detects 17 new memory leaks.

The rest of the paper proceeds as follows. In Section 2 we outline how our tool detects memory errors. In Section 3, 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 4 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 5 we present the experimental evaluation on the real Rust projects. In Section 6 we compare our proposal with the relevant related work, while in Section 7 we conclude and discuss possible future work.

Code and data availability: The code of the tool, the datasets, and the scripts used for our experiments are available on the online repository [21].

2 CREMA at Work

We provide two examples of memory issues, namely a *never-free*, aka *memory leak*, and a *use-after-free*, that arise when we use the unsafe features of Rust carelessly. Moreover, we show how the analysis implemented in CREMA can help discover these issues statically.

First, consider the code of Figure 1, where we allocate a memory cell in the heap through a smart pointer represented by the datatype `Box` (line 4). After converting it to a raw pointer `z_raw` and forgetting its ownership (line 6), we pass `z_raw` to a C function through an FFI call. The C function casts its argument into an integer pointer and frees the corresponding memory. When the Rust code accesses the value pointed by `z_raw` (line 12), the operation produces a *use-after-free* error since the C function has deallocated the memory.

CREMA detects this issue and reports the line of code where the access to the deallocated memory occurs. Intuitively, to uncover this flaw, the analysis first builds the relevant ICFG recording that the control flow passes from Rust code to C code at line 8. Then, the analysis assigns the abstract value $ALLOC$ to the variable `z`, indicating that a cell has been allocated in the heap. Due to the statement `Box::into_raw(z)`, the abstract state is updated to associate `z` with the abstract value MV, thus the ownership of the value is forgotten, and the compiler will not deallocate it when exiting its scope at line 14. The analysis finds then a path in the ICFG where the control reaches a call to the `free` function on that allocated heap cell, causing its deallocation. Moreover, the analysis also discovers that the same memory cell is used again in another Rust statement later on. As a result, a *use-after-free* is detected.

Our second example is taken from the "Napkin Math" repository available on GitHub [42], a project with $4.4K$ stars and 180 forks. This project consists of a command-line tool for computing benchmark performances. A snippet of its code is in Figure 2. CREMA uncovers a memory leak within the `syscall_getrusage` function. This function is a wrapper for invoking the C function `getrusage()` that returns resource usage statistics for the calling process. After user and system CPU time initialization (line 2), the function `syscall_getrusage` allocates on the heap the data structure needed by the function `getrusage()` using a `Box`

```rust
use std::ffi::c_void;
extern "C" {fn cast_and_free_pointer(ptr: *mut c_void);}
fn main() {
    let z = Box::new(90);   // heap allocation: 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 {
        cast_and_free_pointer(z_raw); // pass raw pointer to FFI
    } // 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);
        }}}
```

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

Fig. 1. The Rust code (top) calls a C function (bottom) passing the address of a cell on the heap. The C function frees the memory, causing a use-after-free error in the Rust code.

smart pointer. This smart pointer is then transformed into a raw pointer using a call to `Box::into_raw` (line 10). (Again, the compiler forgets ownership and will not deallocate.) Therefore, the caller is the only one responsible for the memory, which was previously managed by the `Box`. At line 14, the raw pointer is passed to the `libc::getrusage`, which does not free the memory either. Because the raw pointer is never converted back into a `Box` smart pointer to reclaim its ownership, the function `syscall_getrusage` leaks memory at each execution.

Intuitively, CREMA first builds the ICFG; then assigns the value $ALLOC$ to the variable `rusage` and the value MV to the variable `ptr`; finally, the analysis passes `ptr` to the function `libc::getrusage`, abstractly mimicking line 8. Since the analysis finds no path in the ICFG from the allocation point of `rusage` to a call to a `drop` or `free` function, CREMA reports a memory leak.

3 Background

The Rust language. Rust's most distinctive feature is the so-called *ownership* model that enables safe memory management without a garbage collector. It

```
fn syscall_getrusage() {
    let time = libc::timeval { tv_sec: 0, tv_usec: 0, };
    let rusage = Box::new(libc::rusage {  // heap allocation
        ru_utime: time, /* fields population */ ru_nivcsw: 0, });
    let ptr = Box::into_raw(rusage); // ownership forgotten
    let result = benchmark(|| {}, |_test| {
            unsafe {
                // ptr passed to getrusage (C) function
                libc::getrusage(0, ptr);
            }
            true},).unwrap();
    result.print_results("Sycall getrusage(2)", 0);
    // heap memory allocated never freed either in C or in Rust
}
```

Fig. 2. Memory leak Napkin Math

consists of a set of rules that govern how a program manages memory [24], ensuring that 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 structures, functions and threads to share values, 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* [24]. The so-called *borrow checker* statically enforces the above rules. Technically, it compares scopes to determine whether all borrows are valid, and it operates on the Mid-level intermediate representation (MIR), an internal 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 the core Rust MIR considered here. 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 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

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

Constant	z	$\in$	Z
BasicBlock	bb	$::=$	$s.t \mid t$
Operation	$\oplus$	$::=$	$+ \mid \times \mid \ldots$
Operand	o	$::=$	$z \mid x@p$
Lifetime	l	$\in$	$\mathcal{L}$
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	$::=$	$s_1.s_2 \mid \texttt{nop} \mid x@p \leftarrow e \mid \ldots$
Terminator	t	$::=$	$\texttt{call(F, } o_1, \ldots, o_n\texttt{)} \mid x@p \leftarrow \texttt{new} \mid \texttt{drop}(x@p) \mid \ldots$

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 [11]. Abstract interpretation is grounded in lattice theory and Galois connections [41]. 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 that $\alpha(v)$ 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* [33], where each abstract element represents a set of possible concrete states. Abstract operations [34], 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* [8] $f \colon L \to L$, where L is a complete lattice [15]. When L satis-

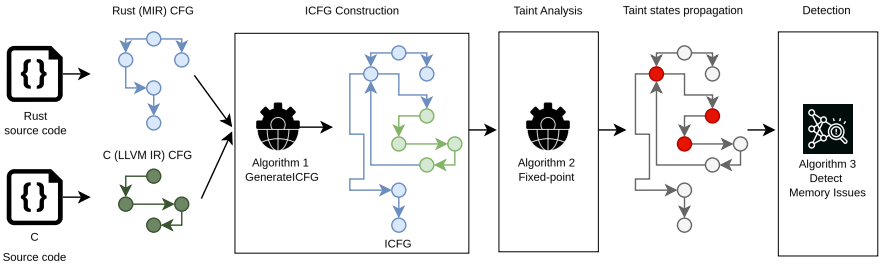

Fig. 3. An overview of our cross-language static taint analysis.

fies the *ascending chain condition* [22], 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 computed anyway through a widening operator [10].

Taint analysis. Taint analysis is a common information-flow analysis [47] that captures mainly the explicit flows. It can be both static and dynamic [4]. The analysis reasons on the control and data dependence arising from a *source statement* to a *sink statement* [49]. The *source-sink* analysis inspects explicit information flow to detect software bugs via value-flow reachability [45]. In brief, the analysis is formalized as follows: let *Var* be the set of all variables of a program and S be the set of all statements. A source is a pair, written $v_{src}@s_{src}$, where s_{src} is the statement that introduces the variable v_{src}. Similarly, a sink is a 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 analyzes the (abstract) effects caused by the current (abstract) execution.

4 Analysis Design

Figure 3 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 we use SVF [45] 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, which discovers those variables, if any, 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. (Actually, a couple of dummy nodes are inserted for a better control of the jump to/return from the foreign code CFG, these nodes store the association between variables of the two IRs). Finally, the algorithm returns the ICFG, which represents the combined control flow of both the Rust and C components.

Algorithm 1 GenerateICFG

```
1: function GenerateICFG(rust_code, c_llvmIR)
2:     ffi_funcs ← getRustFFI(rust_code)
3:     rust_mir ← CompileToMIR(rust_code)
4:     c_cfg ← ParseLLVMIRToCFG(c_llvmIR)
5:     icfg ← rust_cfg
6:     for all block b in rust_cfg.blocks do
7:         t ← b.terminator
8:         if t is Call then
9:             if callee ∈ ffi_funcs then
10:                d_c ← CreateDummyCall(t, callee)
11:                e_in ← getEntryPointBody(c_llvmIR, callee)
12:                AddEdge(icfg, b → d_c → e_in)
13:                e_out ← FindExitPoint(e_in)
14:                d_r ← CreateDummyRet(e_out)
15:                AddEdge(icfg, e_out → d_r → b.return_target)
16:            else
17:                d_c ← CreateDummyCall(t, callee)
18:                d_r ← CreateDummyRet(t)
19:                AddEdge(icfg, b → d_c → d_r → b.return_target)
20:            end if
21:            for all u in t.targets do
22:                AddEdge(icfg, b → u)
23:            end for
24:        end if
25:    end for
26:    return ICFG
27: end function
```

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

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

ordered according to the Hasse diagram of Figure 4 [18]. The bottom element $\bot$ (the minimum element of the lattice) represents the undefined value; the al-

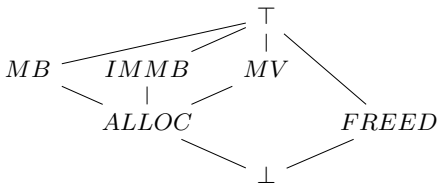

Fig. 4. *Cell_value* partial order.

location 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 $\sigma \colon Var \to Cell\_value$ 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 $\forall \sigma_1, \sigma_2 \in A\_mem, \sigma_1 \sqsubseteq \sigma_2 \iff \forall v \in Var, \sigma_1(v) \sqsubseteq \sigma_1(v)$. Similarly, the join operator $\sqcup$ combines two abstract states $\sigma_1, \sigma_2 \in A\_mem$ in a pointwise manner: $\forall v \in Var\ (\sigma_1 \sqcup \sigma_2)(v) = \sigma_1(v) \sqcup \sigma_2(v)$. 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 $as \colon B \to A\_mem$ 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 $(Abs\_state, \sqsubseteq)$ 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 [50]. 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 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. *MB*). Otherwise, it returns the bottom element to denote that its value concerns no references in the heap. (Also, there are trivial transfer functions for the auxiliary dummy nodes inserted by Algorithm 1.)

We then 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

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

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

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 $[\![\mathrm{o}_i]\!]^a_\sigma$. 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 rely on the results of a taint analysis to detect whether a memory cell is accessed after being freed, freed multiple times, or never freed at all. To compute the taint analysis, we leverage, as usual, a worklist algorithm to iteratively propagate taint information across the ICFG. 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.

The inter-procedural analysis is triggered when a dummy call node in the ICFG is encountered: if the dummy call node is for an internal (Rust) function, the processing order of the ICFG nodes is frozen, and the entry node of the callee

body is visited, as well as its successors. When all callee nodes are completely visited, the previous execution order is resumed (the next node will be the dummy return node). Concerning external function, during the ICFG construction, each FFI invocation is handled by inserting its body between dummy call and dummy return nodes. So, the visit order is the one reflected by the ICFG. The tool also achieved intra-procedural analysis by setting the desired function entry node. Once the fixed point is computed, the memory issues detection phase begins by analyzing the taint (basic block) sources. During the path evaluation, it manages the aliases encountered by inserting them into the allocation set they refer to, relying on a union-find data structure.

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 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 it classifies them into different types of memory-related issues. The algorithm iterates over each path p starting from a basic block b, and examines the taint state of each variable v in b. 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.

5 Implementation and Evaluation

Tool internals. CREMA [21] is mainly implemented in Rust and consists of about 5000 lines of code that exploits the Rust compiler APIs to obtain the HIR and MIR representations of a source file. It also includes a small module written in C++ that analyzes C functions, extracts their LLVM representation, and builds their CFG through the SVF [45] static analysis tool. Given as input a directory containing a Rust project, the tool starts from the program entry point, i.e., from the `main` function, and proceeds as follows. It first invokes the compiler APIs to generate the HIR representation of the code, inspects it to collect the FFI function invocations, and retrieves the corresponding CFGs; it then extracts the corresponding MIR and builds the ICFG implementing Algorithm 1. In addition,

Algorithm 2 Fixed-point algorithm(ICFG)

```
1: function FIXED_POINT(ICFG)
2:     AbstState ← Dict()                          ▷ Contains σ_b ∈ A_mem for each block b
3:     TaintStates ← Dict()                        ▷ Contains the taint of each variable in each block
4:     Worklist ← Queue(main)                      ▷ Initialize the Worklist with the main block
5:     for each basic block b in ICFG do
6:         for each variable v in b do
7:             AbstState[b][v] ← ⊥                 ▷ Initialize abstract states
8:             TaintStates[b][v] ← ∅               ▷ Initialize taint states as empty set
9:         end for
10:    end for
11:    Worklist.enqueue(entry_node)
12:    while Worklist is not empty do
13:        curr_node ← Worklist.dequeue()
14:        for each succ_node in curr_node.successors do
15:            for each variable v in succ_node do
16:                new_abst_state ← S⟦curr_node⟧^a AbstState  ▷ where S is the code of succ_node
17:                if new_abst_state ≠ AbstState[curr_node][v] then
18:                    AbstrStates[succ_node][v] ← new_abst_state
19:                end if
20:                new_taint_st ← UPDATE_TAINT_STATE(v, curr_node, TaintStates, AbstState)
21:                if nw_taint_st ≠ TaintStates[curr_node][v] then
22:                    TaintStates[succ_node][v] ← new_taint_st
23:                end if
24:            end for
25:            Worklist.enqueue(succ_node)
26:        end for
27:    end while
28:    return DETECT_MEM_ISSUES(ICFG, TaintStates)
29: end function
```

CREMA can be invoked to analyze a specific function by just passing its name, besides considering a whole Rust project. In practice, the analysis proceeds by applying Algorithms 2 and 3, described previously, with some additional mechanism to deal with aliasing. More precisely, the implementation keeps track of memory cell aliases by inserting them into the same allocation set they refer to.

Once the analysis is performed, CREMA outputs a report indicating possible memory errors affecting the analyzed Rust project together with a `graphviz` [19] representation of the ICFG, where the nodes are colored according to the memory operations performed therein. Besides the standard memory errors, CREMA also generates a warning when a Rust heap allocation is freed by the FFI code. This is because Rust code and C code may use different memory allocator implementations, e.g., jemalloc and libc malloc. Mixing memory allocators is considered undefined behavior [27], so CREMA raises a warning.

Tool evaluation. We evaluate CREMA effectiveness by answering the following research questions:

RQ1 Can CREMA detect common memory errors and with what precision?
RQ2 Can CREMA detect memory issues in real-world codebases?

To reply to **RQ1**, we prepare a dataset consisting of 73 Rust tests, which present 68 memory issues broken down as follows: 22 *never frees/memory leaks*; 26 *double frees*; and 20 *use-after-frees/undefined behaviors*. Of these issues, 56 occurred in pure Rust code and 12 in Rust interacting with C FFI. These tests

Algorithm 3 Detect memory issues

```
1: function DETECT_MEM_ISSUES(ICFG, TaintStates)
2:     use-after-free ← MultiSet()
3:     double-free ← MultiSet()
4:     never-free ← MultiSet()
5:     for each path p starting from block b in icfg do
6:         for each variable v in TaintStates[b] do
7:             if ∃b′ ≠ b in p s.t. TaintStates[b][v] ∩ TaintStates[b′][v] ⊇ {free} then
8:                 double_free.add(v)
9:             else if TaintStates[b][v] ≠ ∅ ∧ ∀b′ ≠ b in p free ∉ TaintStates[b′][v] then
10:                never_free.add(v)
11:            else if TaintStates[b][v] ≠ ∅ ∧ ∃b′ ≠ b in p use ∈ TaintStates[b′][v] then
12:                use_after_free.add(v)
13:            end if
14:        end for
15:    end for
16:    return use_after_free, double_free, never_free
17: end function
```

are especially designed to uncover typical weaknesses, aiming to stress the tool on corner cases leading to over-approximation. Using this dataset, we perform a quantitative assessment using standard classification metrics: *Precision*, *Recall*, *Accuracy*, *F1-score*, *Specificity*, and *False Negative Rate*. These metrics are in Table 3 and provide a comprehensive overview of CREMA's detection capability and its tendency to produce false alarms. The table uses the following notation: TP (True Positive) denotes the number of memory errors correctly reported; FP (False Positive) is the number of non-existent errors incorrectly reported; FN (False Negative) means the number of actual memory errors that the tool failed to detect; and TN (True Negative) denotes the number of correct identifications of safe code with no errors. Recall that our analysis is sound, thus our tool reports no false negatives (FN is always 0), which are then not displayed in the table. The row *Precision* displays the proportion of reported memory errors that are truly erroneous: CREMA correctly reports an issue 97.14% of times and generates a false positive only in 2, 86% of the reported notices. Of course, false positives are unavoidable because of over-approximation, typically spawn when processing a branching node, e.g., see the code of `cstr_cargo_df_ffi` in the repository. The *Accuracy* measures all correct reports (memory error found or no report) over

Table 3. CREMA performances over 73 Rust test projects with 68 memory issues. `FP`: False Positives, `FN`: False Negatives, `TP`: True Positives, `TN`: True Negatives.

Metric	Formula	Value
Precision	$\frac{TP}{TP+FP}$	$97,14\,\%$
Recall	$\frac{TP}{TP+FN}$	$100\,\%$
Accuracy	$\frac{TP+TN}{TP+TN+FP+FN}$	$97,5\,\%$
F1-score	$\frac{TP}{TP+0.5(FP+FN)}$	$98{,}55\,\%$

Table 4. Memory issues reported by CREMA on analyzed Rust projects.

Project Name	Reported Issue	Alarms	False Alarms	Actual Issues	Execution Time
noGenerator [36]	Memory leak	2	1	1	1.551s
wasm-demo [14]	Memory leak	1	–	1	1.573s
skip-list-test [3]	Memory leak	10	–	10	1.661s
rusant [23]	Memory leak	1	–	1	1.673s
napkin-math [42]	Memory leak	1	–	1	1.491s
shared-register [32]	Memory-leak	1	–	1	3.099s
shared-register [32]	Double-free	1	1	–	3.099s
whisper-rs-example [9]	Memory leak	2	1	1	2.078s
lock-free [17]	Memory leak	1	–	1	1.561s
rust-memory [48]	Memory leak	4	4	–	0.608s
unsized_struct [31]	Memory leak	1	1	–	1.497s

CREMA reported no alarm for the following projects: rust-boks [7]. unsafely-created-owned-type [28], lock_free_non_blocking_linked_list [40], c-callback-rust-closure [52], concurrent-verification [43], stackswap-coroutines [37], rc-playground [35], square [20], rust_hw3 [38], openapi-client-gen [5].

total test cases: CREMA correctly classifies $97,5\%$ of cases. *F1-score* estimates the predictive performance between *Precision* and *Recall* as a harmonic mean. In summary, our answer to **RQ1** is positive: CREMA successfully detects common memory issues with a good true detection rate, as Table 3 shows.

To answer **RQ2**, we implement an automated script (available in the online repository [21]) to search on GitHub Rust projects that use potentially unsafe operations (e.g., `into_raw`, `mem::forget`, `mem::drop`) in their code. We evaluated CREMA on 20 projects found by our script, which range from concurrent data structures to benchmarking and testing libraries. Table 4 reports the *names* of the analyzed projects, the *type* of memory issues discovered, the *total number* of *alarms* raised, the count of *false alarms*, the number of *actual memory issues* discovered and the *execution time* required by the analysis. CREMA detected 17 new *memory leaks* on 8 projects, and each project was analyzed in just a few seconds. Note that false positives concern heap allocation aliases. Moreover, some of the analyzed projects depend on external code implemented in languages other than C, e.g., napkin-math [42] and rusant [23]. For this reason, we analyzed the relevant portions of code addressed by CREMA. Other projects, such as wasm-demo [14], have no main entry point; thus, in this case, we exploit the CREMA feature of starting the analysis from a specified function. Our repository includes the code of the analyzed projects and a detailed report of the memory errors discovered, and how to replicate the results. In summary, our answer to **RQ2** is positive: CREMA was able to analyze 20 Rust projects from GitHub and to detect 17 new memory leaks, as reported in Table 4.

6 Related Work

In the literature, several studies use MIR to detect potential vulnerabilities in Rust code. Miri [46] is an interpreter for Rust MIR focused on identifying undefined behavior by running binaries and test suites of cargo projects. The abstract

interpreter MIRAI [46] statically verifies assertions inside the code and identifies possible runtime panics. MirChecker [26] extends MIRAI to combine numerical static analysis with symbolic execution for identifying runtime panics and lifetime corruptions. Another proposal is SafeDrop [12] 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 caused by unsafe code. Instead, RCanary [13] defines a model checker to detect leaks across the semi-automated boundary within Rust programs, implemented as an external component in Cargo. The main difference between ours and the proposals above is that CREMA deals with FFI functions, and analyzes both Rust and C code to identify memory errors.

The proposal most similar to ours is FFIChecker [27]. It relies on an abstract domain, transfer functions, and a fixed-point algorithm, though applied differently. 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 distinguish mutable from immutable borrows. In contrast, our model keeps them apart, thus providing a more accurate tracking of abstract states of heap-memory objects and improving the analysis results. By capturing more detailed insights, we facilitate finding the root cause of an error and 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 signature and behavior, we map functions to their corresponding types and behavior. This modular approach enables 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, as shown by the experimental comparison reported in the online repository [21].

The comparison was only carried out on Rust–C FFI programs because FFIChecker does not analyze pure Rust programs, although forgotten ownership can cause memory leaks. On the cases considered CREMA reports fewer spurious alarms and is then more precise in detecting potential memory errors than FFIChecker. Also CREMA is consistently faster than FFIChecker by a few seconds on the same Rust programs analyzed, in spite of their reduced size. This makes us confident that our tool offers a practical performance advantage.

7 Conclusion

We proposed a static taint analysis that targets Rust programs using unsafe blocks and foreign code through the FFI mechanism. Here we tailored our analysis on C as an external language and detected the common memory errors use-after-free, never-free, and double-free. However, any language that compiles to LLVM IR can be taken instead. 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 available. Our analysis relies on an abstract domain designed to track taint information related to heap management, fo-

cusing 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 language semantics. Finally, we presented a fixed-point algorithm that propagates the taint information across the ICFG, then used by a checking algorithm to detect possible memory issues. We implemented our analysis in the tool CREMA and tested its efficacy on two datasets of Rust programs. The first contains 73 test cases designed to test the precision of the analysis. The second consists of about 20 projects from GitHub to show effectiveness in finding bugs in the field. In the real cases, CREMA detected 17 new memory errors that may be detrimental to program executions. Although in a preliminary version, CREMA performs reasonably well also on projects of non-negligible size.

There are several future research directions. First, to make the analysis more precise, we plan to extend the abstract domain and consider explicitly annotated lifetimes, which requires adapting transfer functions. As for the C side, before the creation of the ICFG, we will improve the precision of the analysis by computing points-to information directly on the LLVM IR. Including a module responsible for pointer/alias analysis will also improve precision and impact on the scalability. For that, we will consider two main approaches. Andresen's pointer analysis [6] 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 [44] offers a less precise but more time-efficient analysis, which is almost linear in the number of program statements. Finally, we plan to make our approach beneficial to build secure multi-language code, not only concerning more memory errors, e.g., *use-before-initialization,* but also for all the other security challenges arising from all unsafe Rust features.

Acknowledgments. Work supported by projects SERICS (PE00000014) and PRIN PNRR 2022 AM∀DEUS (P2022EPPHM) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

References

1. The LLVM compiler infrastructure, https://llvm.org/
2. The Rustonomicon, https://doc.rust-lang.org/nomicon/
3. Abhijeetbhagat: skip-lest-test, https://github.com/abhijeetbhagat/skip-list-test
4. Aggarwal, A., Jalote, P.: Integrating static and dynamic analysis for detecting vulnerabilities. In: COMPSAC (1). pp. 343–350. IEEE Computer Society (2006)
5. Alexander Schütz: openapi-client-gen, https://github.com/AlexanderSchuetz97/openapi-client-gen
6. Andersen, L.O.: Program analysis and specialization for the c programming language (1994)
7. Anookinov-training-ground: rust-boks, https://github.com/anookinov-training-ground/rust-boks

8. Baladi, V., Keller, G.: Zeta functions and transfer operators for piecewise monotone transformations. Communications in mathematical physics **127**, 459–477 (1990)
9. Bruceunx: whisper-rs-example, https://github.com/bruceunx/whisper-rs-example
10. Cortesi, A.: Widening operators for abstract interpretation. In: SEFM. pp. 31–40. IEEE Computer Society (2008)
11. Cousot, P., Cousot, R.: Abstract interpretation frameworks. J. Log. Comput. **2**(4), 511–547 (1992), https://doi.org/10.1093/logcom/2.4.511
12. Cui, M., Chen, C., Xu, H., Zhou, Y.: Safedrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis. ACM Trans. Softw. Eng. Methodol. **32**(4), 82:1–82:21 (2023)
13. Cui, M., Xu, H., Tian, H., Zhou, Y.: rcanary: Detecting memory leaks across semi-automated memory management boundary in rust. IEEE Trans. Software Eng. **50**(9), 2472–2484 (2024)
14. Dan Paz-Soldan: wasm-demo, https://github.com/danpaz/wasm-demo
15. Davis, A.C.: A characterization of complete lattices. Pacific J. Math **5**(2), 311–319 (1955)
16. Dmitry, B.: How much Rust in Firefox?, https://4e6.github.io/firefox-lang-stats/
17. DorukCem: lock-free, https://github.com/DorukCem/lock-free
18. Eklund, P.W., Ducrou, J., Brawn, P.: Concept lattices for information visualization: Can novices read line-diagrams? In: ICFCA. Lecture Notes in Computer Science, vol. 2961, pp. 57–73. Springer (2004)
19. Ellson, J., Gansner, E.R., Koutsofios, E., North, S.C., Woodhull, G.: Graphviz - open source graph drawing tools. In: GD. Lecture Notes in Computer Science, vol. 2265, pp. 483–484. Springer (2001)
20. EverSeenTOTOTO: square, https://github.com/EverSeenTOTOTO/square
21. Franceschi, A., Galletta, L., Degano, P.: Detecting Memory Errors in Rust Programs Including Unsafe Foreign Code (Supplementary Material) (2025), https://github.com/AFx3/crema-static-analyzer, accessed on June 2025
22. Hazewinkel, M., Gubareni, N., Kirichenko, V.V.: Algebras, Rings and Modules: Volume 1, vol. 575. Springer Science & Business Media (2006)
23. JakWai01: rusant, https://github.com/JakWai01/rusant
24. Klabnik, S., Nichols, C.: The Rust programming language. No Starch Press (2023)
25. Li, H., Guo, L., Yang, Y., Wang, S., Xu, M.: An empirical study of rust-for-linux: The success, dissatisfaction, and compromise. In: USENIX ATC. pp. 425–443. USENIX Association (2024)
26. Li, Z., Wang, J., Sun, M., Lui, J.C.S.: Mirchecker: Detecting bugs in rust programs via static analysis. In: CCS. pp. 2183–2196. ACM (2021)
27. Li, Z., Wang, J., Sun, M., Lui, J.C.S.: Detecting cross-language memory management issues in rust. In: ESORICS (3). Lecture Notes in Computer Science, vol. 13556, pp. 680–700. Springer (2022)
28. Luiz' Labs: unsafely-created-owned-type, https://github.com/lffg-labs/unsafely-created-owned-type
29. Matsakis, N.D., II, F.S.K.: The rust language. In: HILT. pp. 103–104. ACM (2014)
30. Mergendahl, S., Burow, N., Okhravi, H.: Cross-language attacks. In: NDSS. The Internet Society (2022)
31. MiguelX413: unsized_struct, https://github.com/MiguelX413/unsized_struct
32. Namra Patel: shared-register, https://github.com/namrapatel/shared-register

33. Nielson, F., Jones, N.: Abstract interpretation: a semantics-based tool for program analysis. Handbook of logic in computer science **4**, 527–636 (1994)
34. Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis. springer (2015)
35. Oleksii Halahan: rc-playground, https://github.com/skyne98/rc-playground
36. Pamquale: Nogenerator, https://github.com/pamquale/noGenerator
37. Phase: stackswap-coroutines, https://github.com/phase/stackswap-coroutines
38. PM25000: rust_hw3, https://github.com/PM25000/rust_hw3
39. Rust compiler development guide - the MIR (Mid-level IR), https://rustc-dev-guide.rust-lang.org/mir/index.html
40. Santo Shakil: lock_free_non_blocking_linked_list, https://github.com/santoshakil/lock_free_non_blocking_linked_list
41. Shmuely, Z.: The structure of Galois connections. Pacific Journal of Mathematics **54**(2), 209–225 (1974)
42. Simon Eskildsen: napkin-math, https://github.com/sirupsen/napkin-math/
43. Singh, S.: concurrent-verification, https://github.com/sujalsin/concurrent-verification
44. Steensgaard, B.: Points-to analysis in almost linear time. In: POPL. pp. 32–41. ACM Press (1996)
45. Sui, Y., Xue, J.: SVF: interprocedural static value-flow analysis in LLVM. In: CC. pp. 265–266. ACM (2016)
46. The Rust Programming Language (Community): Miri, https://github.com/rust-lang/miri
47. Tripp, O., Pistoia, M., Fink, S.J., Sridharan, M., Weisman, O.: TAJ: effective taint analysis of web applications. In: PLDI. pp. 87–97. ACM (2009)
48. Vassjozsef: rust-memory, https://github.com/vassjozsef/rust_memory
49. Wang, T., He, H., Liu, X., Li, S., Jia, Z., Jiang, Y., Liao, Q., Li, W.: Conftainter: Static taint analysis for configuration options. In: ASE. pp. 1640–1651. IEEE (2023)
50. Wilhelm, R.: Principles of abstract interpretation: *By Patrick Cousot* MIT press, 2021, ISBN 9780262044905, pp. 1-819. reviewed by reinhard wilhelm. Formal Aspects Comput. **34**(2), 1–3 (2022)
51. Xu, H., Chen, Z., Sun, M., Zhou, Y., Lyu, M.R.: Memory-safety challenge considered solved? an in-depth study with all rust cves. ACM Trans. Softw. Eng. Methodol. **31**(1), 3:1–3:25 (2022)
52. Yiannis Marangos: c-callback-rust-closure, https://github.com/oblique/c-callback-rust-closure