RUSTY: Effectively Detecting Multilingual Rust Memory Bugs with Interprocedural Static Analysis

Mingliang Liu Baojian Hua*
School of Software Engineering, University of Science and Technology of China
Suzhou Institute for Advanced Research, University of Science and Technology of China
liumingliang@mail.ustc.edu.cn bjhua@ustc.edu.cn

***Abstract*—Rust has emerged as a highly promising systems programming language for security-critical applications, owing to its strong type system and ownership model, which effectively prevent memory safety issues. However, real-world Rust applications are often multilingual and must interact with external modules written in memory-unsafe languages (*e.g.*, C/C++) through foreign function interface (FFI). These interactions require integrating disparate memory management mechanisms and typically bypass Rust's compiler safety checks, making them highly error-prone. Consequently, this may introduce memory safety issues, thereby compromising Rust's memory safety guarantees.**

In this paper, we propose RUSTY, a technique that enables cross-language static analysis to effectively detect memory safety bugs in multilingual Rust programs. Specifically, we first convert Rust and other languages into a unified intermediate representation to overcome language heterogeneity. Second, we employ value-flow analysis to perform program slicing, establishing a precise analysis scope. Building upon these foundations, we develop context-sensitive interprocedural abstract interpretation combined with points-to analysis to effectively identify memory safety bugs in multilingual interactions. We implemented a prototype of RUSTY and conducted extensive experiments to evaluate its practical effectiveness and performance using both micro-benchmarks and real-world CWEs. The experimental results demonstrate that RUSTY can effectively detect memory bugs in multilingual Rust programs while maintaining favorable detection efficiency and resource overhead.

***Index Terms*—Static Analysis, Rust, Multilingual**

I. INTRODUCTION

Rust has emerged as a highly promising system programming language for security-critical applications, owing to its unique ownership model and strong type system offering proven effectiveness in preventing memory safety issues [1] [2]. These advantages have positioned Rust as a compelling alternative to C/C++, leading to its successful adoption in security-critical domains including operating systems [3] [4] [5] [6], databases [7], web browsers [8], and language runtimes [9]. In the coming decade, a desire to secure software infrastructure without sacrificing efficiency will make Rust a more promising language.

Unfortunately, Rust's memory safety guarantee is not a panacea and is threatened by ever-increasing multilingual programming in Rust [10] [11]. Generally, many real-world Rust applications (*e.g.*, Linux kernel [4] and Firefox [12]) are multilingual to interact with external modules written in memory-unsafe languages (*e.g.*, C/C++) via foreign function interfaces (FFIs). However, while Rust's multilingual programming paradigm brings significant benefits including legacy code reuse, toolchain flexibility, and extreme performance, it introduces new attack vectors to Rust applications. Specifically, vulnerabilities in the external modules developed by unsafe languages like C may bypass the safety checks of Rust [1] [10] [13] [14], thereby undermining the security guarantee of Rust. More concerning, even the modules in different languages are secure, vulnerabilities may still manifest on and across language boundaries [15] [1] [16] [13], due to language disparities including memory models, memory management mechanisms, and data representation strategies, among others. Therefore, detecting memory bugs of multilingual programming Rust is both critical and urgent.

Recognizing this need, researchers have conducted preliminary studies for analyzing multilingual Rust programs [17] [18] [11]. Unfortunately, despite this promising progress, the research field of multilingual Rust program analysis remains largely underexplored due to several challenges to be tackled. First, some approaches are limited to particular scenarios, making it challenging to address more complex multilingual interactions arising in real-world applications. For example, the state-of-the-art multilingual memory issue detector, FFIChecker [18], only tracks heap memory allocated in Rust and passed to C/C++, overlooking heap objects allocated in C/C++ and passed to Rust. Therefore, FFIChecker struggles to detect memory issues caused by C/C++ objects. Second, adapting approaches developed for other languages [19] [20] [21] to Rust remains challenging or even impossible, due to the dramatic language feature discrepancies between Rust and other languages. For example, Pungi [19] detects reference-counting memory errors in Python/C multilingual programs by transforming them into affine representations, but adapting this approach to Rust/C remains difficult because Rust leverages an explicit ownership-based memory management rather than Python's reference counting-based garbage collection.

In this paper, to fill this gap, we take a new step towards proposing a comprehensive analysis to detecting memory bugs in multilingual Rust programs. Our key insight is that to analyze multilingual programs, it is essential to treat Rust and external languages such as C/C++ uniformly, thereby enabling

* The corresponding author.

a holistic analysis by integrating their code. However, developing a holistic analysis for multilingual Rust programs needs to tackle three major challenges: **C1**: Language disparity. The heterogeneous languages have different semantics and features (*e.g.*, the unique ownership mechanisms of Rust). Thus, the holistic analysis should incorporate these distinct language characteristics. **C2**: Analysis cost-effectiveness. The holistic analysis on multilingual Rust programs tends to process more code than that in single-language. Therefore, to be practically useful, the analysis should be cost-effective to scale to real-world applications. **C3**: Coverage trade-off. To uncover all potential bugs, the analysis should cover all feasible execution paths, while ensuring the reasonable analysis time for practicality.

In this paper, following the aforementioned insight of holistic analysis, we present RUSTY, a novel approach that enables holistic static analysis to effectively detect memory safety bugs in multilingual Rust programs while addressing the aforementioned challenges. Specifically, to address the challenge **C1** of language disparity, we propose to leverage the LLVM IR, an intermediate language that is essential to compile many languages, to represent Rust and C/C++ uniformly, thereby overcoming the challenge of language heterogeneity. While LLVM IR has shown paramount success in analyzing many single languages [22] [23] [24], our work is the first to leverage LLVM IR for Rust multilingual holistic analysis. Our use of LLVM as the analysis backbone enables us to easily unify most language characteristics and only needs few supplements of language-specific features, including FFI information and ownership semantics that are essential to the analysis.

To address the challenge **C2** of analysis cost-effectiveness, we design a new value-flow analysis to perform program slicing on multilingual programs, which traces the propagation paths of multilingual objects at and across language boundaries. Specifically, we trace value-flow edges by annotating relevant functions along these paths in different languages. As a result, this tracing process establishes a precise analysis scope for subsequent phases, thereby enhancing analysis cost-efficiency.

To address the challenge **C3** of coverage trade-off, we leverage a static program analysis approach instead of a dynamic one to detect memory issues in the program slice. Specifically, we design a context-sensitive interprocedural abstract interpretation [25] [26] supplemented by a points-to analysis [27] [28] on the multilingual programs. The abstract interpretation approximates actual program execution by defining dedicated abstract domains and transfer functions. To this end, we propose an abstract domain for our analysis that captures the allocation and deallocation states of memory objects within the program. Furthermore, we define transfer functions for each statement to update and propagate the abstract states of memory objects. Building upon these, we leverage a fixed-point algorithm [29] to abstractly execute the program. Finally, we introduce a set of detection rules to identify memory issues by verifying the abstract states of memory objects at critical program points.

To validate our approach, we implement a prototype for RUSTY, and conduct extensive experiments to evaluate its effectiveness and performance. We select C/C++ as the target unsafe language since Rust often cooperates with C/C++ in multilingual programming, but our approach is generic and can be readily extended to other language combinations. Our evaluations are performed on two datasets: 1) a micro-benchmark comprising common memory bug patterns caused by the interaction of Rust and C/C++; and 2) real-world CWEs [30] [31]. The evaluation results demonstrate that RUSTY can effectively detect multilingual memory bugs with a true positive rate of 76% on real-world CWEs, outperforming existing approaches. Meanwhile, RUSTY maintains favorable detection efficiency and resource overhead.

In summary, this paper makes the following contributions:

- We propose a holistic analysis approach to effectively detect memory issues in multilingual Rust programs.
- We design and implement a software prototype for RUSTY to validate our approach.
- We conduct a comprehensive evaluation to demonstrate that our approach is effective and efficient, outperforming state-of-the-art approaches. The source code is publicly available to facilitate further research[1].

The rest of this paper is organized as follows. Section II introduces the background and the motivation of this work. Section III and Section IV present our approach. Section VI presents the experimental evaluation of RUSTY. Section VII discusses limitations and future directions. Section VIII reviews related work, and Section IX concludes.

II. BACKGROUND AND MOTIVATION

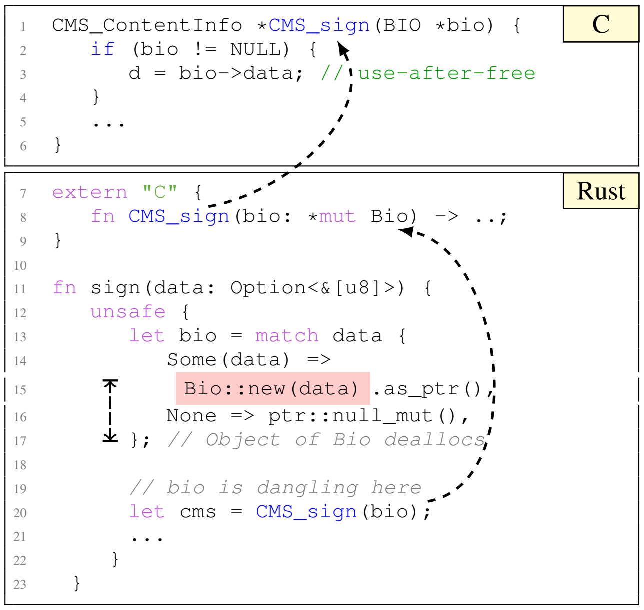
In this section, we concisely explain the concepts of Rust multilingual programming and our motivations that can aid in understanding this paper, and then give the threat model of our work.

A. *Rust Multilingual Programming*

Rust is a highly promising system programming language that can effectively eliminate common memory safety issues in traditional unsafe languages without compromising performance. As a result, some widely used software infrastructures are switching to Rust for developing their components to enhance their own security [3]–[9]. Nevertheless, limited by legacy code reuse, toolchain limitations and other challenges of reality, Rust often needs to interact with existing unsafe code. These interactions require developers to write code to bridge Rust and other languages, referred to as foreign function interface (FFI) [32].

FFI is a mechanism that enables interoperability between Rust and other programming languages. Through FFI, Rust can both invoke functions implemented in other languages and expose its own functions to external languages. Rust declares FFI using an `extern` block, where the calling convention can also be explicitly specified. For example, as shown in

[1]https://doi.org/10.5281/zenodo.16677331



```
1  CMS_ContentInfo *CMS_sign(BIO *bio) {
2     if (bio != NULL) {
3        d = bio->data; // use-after-free
4     }
5     ...
6  }
```

```
7  extern "C" {
8     fn CMS_sign(bio: *mut Bio) -> ..;
9  }
10
11 fn sign(data: Option<&[u8]>) {
12    unsafe {
13       let bio = match data {
14          Some(data) =>
15            Bio::new(data).as_ptr(),
16          None => ptr::null_mut(),
17       }; // Object of Bio deallocs
18
19       // bio is dangling here
20       let cms = CMS_sign(bio);
21       ...
22     }
23   }
```

Fig. 1: An example vulnerability that we adapted from CVE-2018-20997 [33], which is uncovered in the openssl Rust crate with a critical CVSS score of 9.8. This vulnerability incurs a use-after-free due to improper multilingual interaction.

line 7 of Fig. 1, this syntax declares a C function defined in line 1 and indicates the use of the C ABI. Conversely, Rust can also export its functions to other languages through the extern keyword. FFI operations are inherently unsafe as the Rust compiler cannot verify the safety of external code. Consequently, all FFI usage must be explicitly enclosed in an **unsafe** block, as demonstrated in line 12.

B. Motivation

Our work is motivated by the recently revealed issues about Rust FFI security [11] [18] [34]. Specifically, while Rust FFI is widely used among the Rust community [35] [36] [37], it constitutes a significant number of issues [15] [34] [38]. Rust FFI-based multilingual programming in Rust is inherently prone to errors, due to two key challenges. First, the unsafe code in the multilingual interaction bypasses Rust compiler's memory safety checks, so its safety guarantees must be manually ensured by the developers. For example, the raw pointer arguments passed to FFI will not be checked by Rust compiler, thus the correctness of its lifetime must be manually guaranteed by developers, otherwise it will incur memory safety issues including null pointer dereferences, use-after-free, double frees, among others. Second, language disparities introduce significant challenges. Specifically, FFI requires the integration of distinct memory models and management paradigms. For instance, Rust employs smart pointers based on the ownership model for automatic compile-time memory management, whereas C relies on explicit manual memory management. Any oversight of these mechanisms may exacerbate memory safety issues.

To put the above discussion into perspective, we present in Fig. 1 a code snippet that we adapted from CVE-2018-20997 [33], a vulnerability uncovered in the openssl Rust crate with a critical CVSS score of 9.8. This vulnerability stems from an oversight of Rust ownership mechanism. Specifically, the foreign function CMS_sign takes as its argument a raw pointer bio, which enforces the conversion of the Rust object data into raw pointers bio (line 13) by using the as_ptr method prior to FFI invocation (line 15). However, the raw pointer bio bypasses Rust's safety checks because it resides in an unsafe block (between line 12 and 22), thus the newly created object bio is automatically dropped without any warnings when exiting the lexical scope of the match block (line 17), rendering bio a dangling pointer after line 17. Consequently, when the dangling pointer bio is passed to the CMS_sign function (line 8 and again 1), it leads to a use-after-free (UAF) vulnerability when the pointer is dereferenced (line 3). This CVE demonstrates that in multilingual interactions, the absence of compiler safety checks coupled with overlooking the language security mechanisms result in severe memory safety issues.

C. Threat Model

Our work focuses on the study of a holistic program analysis for Rust multilingual applications. Therefore, we make the following assumptions in the threat model.

First, we assume that the host environment of Rust applications is trusted. This encompasses the underlying hardware, operating system, compiler, linker, and other components of the Rust toolchain. The security of operating systems, compilers and others is orthogonal to our work, and our assumption can be guaranteed by extensive prior research in these domains [6] [39] [40].

Second, we assume that pure Rust code, including all unsafe code except that related to FFI, is memory-safe and poses no security threats. Given the substantial body of research on pure Rust and unsafe Rust without FFI calls [23] [41] [42], this assumption is reasonable.

Third, we assume all external functions accepting pointer arguments from Rust to be untrusted. For example, if a C/C++ function accepts a Rust object via a pointer argument, it may arbitrarily corrupt this object, thus undermining Rust's safety guarantees.

III. Approach

In this section, we first describe an overview of our approach (§ III-A), information and IR generation (§ III-B), and program slicing (§ III-C). And in the next section (§ IV), we present the abstract interpretation-based static analysis for bug detection.

A. Overview

Fig. 2 presents an overview of the architecture of our approach. It accepts Rust and C/C++ source code as input and generates bug warnings for potential multilingual memory bugs, including their precise bug types and locations in the source code. Rusty consists of four key components, which

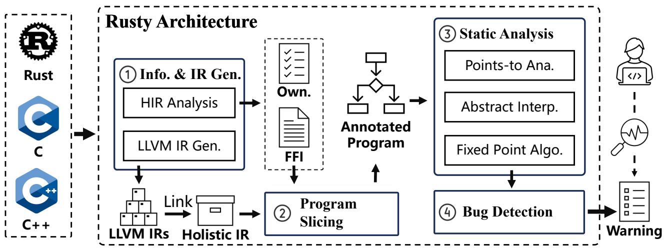


Fig. 2: An overview of RUSTY's workflow.

operate as follows: First, the information and IR generation step (①) takes the source codes as input and extracts the information required for subsequent analysis steps within the compilation pipeline, including the LLVM IR of both Rust and C/C++ files, as well as FFI signatures and ownership information of function parameters derived from the Rust high-level intermediate representation (HIR). Next, program slicing (②) utilizes this information and applies a value-flow analysis to perform program slicing, generating the annotated programs and aims to precisely narrow down the analysis scope. Third, static analysis (③) phase employs the classic approach of abstract interpretation and leverages a fixed-point algorithm to compute the abstract state of the multilingual memory objects at each program point. We will also integrate an interprocedural points-to analysis in this step to accurately determine the points-to set of each pointer. Finally, bug detection component (④) examines object's state at critical program point to identify potential memory bugs based on the detection rules we proposed, and produces the corresponding bug warnings. Subsequently, developers can leverage these diagnostics to inspect the source code and resolve the identified errors.

In the following subsections, we will expand first two steps by explaining more details, and we will describe the last two steps in Section IV.

B. Information and IR Generation

This step generates the necessary information from the compilation pipeline, including the FFI signatures, function parameters' ownership and the LLVM IR to be analyzed. We mainly adopt the following two measures to implement the generation work.

Rust HIR Analysis. We acquire the essential FFI signatures and function parameters' ownership information by analyzing Rust's HIR. The HIR is selected for analysis because it provides not only a simplified structure but also contains crucial ownership information, particularly the unsafe block required for FFI identification. These features absent in both the abstract syntax tree and mid-level intermediate representation. By examining function definitions, we classify a function as an FFI if it satisfies either of the following conditions: 1) it is declared within an `extern` block utilizing the C ABI, or 2) it is a public function employing the C ABI without name mangling. Furthermore, to obtain the parameters' ownership, we analyze composite type parameters in functions and record whether they are passed by reference or by move. We record this information to facilitate the identification of objects that require to be deallocated after the function returns, and we will describe this at Section IV-D.

LLVM IR Generation. RUSTY generates LLVM IR by intercepting the compilation pipeline and injecting additional compilation flags. To enable holistic analysis, we link all IRs into a unified module prior to analysis, thereby ensuring the complete mapping from function declarations to their definition. Furthermore, we apply a `mem2reg` optimization pass [43] to the generated IR, This optimization simultaneously reduces program size and eliminates address-taken pointers that are typically challenging for points-to analysis, while preserving a direct mapping from IR to source code that enables precise source location tracking.

C. Program Slicing

The program slicing step is introduced to refine the programs for analysis, thereby reducing the workload of subsequent static analysis and improving overall detection efficiency. Since we focus on detecting multilingual memory bugs, our idea is that only memory objects crossing the boundary of multilingual interaction can potentially cause multilingual memory safety bugs. Furthermore, the problematic locations are constrained to the propagation paths of these multilingual objects, such as the paths from their allocation sites to all use sites and deallocation sites. Based on this idea, we identify and annotate all paths that propagate multilingual objects in the program, ultimately generating an annotated program.

Our program slicing technique is based on the value-flow analysis approach [44] [45]. This approach traces the propagation paths of variable values through various program operations (*e.g.*, assignments, function calls and load/store operations), and constructs a value-flow graph (VFG) to model these behaviors. The VFG is a directed graph where nodes represent statements, parameter passing or return values, and edges denote the value-flow relationships between variables. Unlike def-use chains that link definition sites and use sites, value-flow edges directly connect two definitions [24], representing the propagation of values from the source definition to the destination definition, as illustrated in Fig. 3b. We complete the construction of the annotated program through two traversal passes over the VFG: a backward traversal and a forward traversal.

Backward Traversal. The backward traversal identifies all paths leading to the allocation sites of multilingual memory objects. Since a memory object can only cross language boundaries through raw pointer variables in multilingual interactions, this traversal initiates from FFI call sites that involve pointer arguments or pointer return values. It then traces backward along value-flow edges of these pointer until locate corresponding memory allocation sites, while annotating all encountered function in the process. As depicted in line 9 of Algorithm 1, the backward traversal algorithm is implemented as a worklist algorithm. It first initializes worklist W with all pointer operands of FFI calls, including both pointer arguments

Algorithm 1: Program Slicing Algorithm.

```
   Input: The program $P$ and its value-flow graph $G$
   Output: The annotated program $P$ and the entry
           function set $F$
1  Function ProgramSlicing($P, G$):
2      $P$ = BackwardTraversal($P, G$)
3      $P$ = ForwardTraversal($P, G$)
4      $F \leftarrow \{\}$
5      for each annotated function $f \in P$ do
6          if no caller of $f$ is be annotated then
7              $F = F \cup f$
8      return $P, F$
9  Function BackwardTraversal($P, G$):
10     $W \leftarrow \{\}$
11     for each pointer operand $p \in$ FFI call do
12         $W$.append($p$)
13     while $W \neq \emptyset$ do
14         $p \leftarrow W$.removeNext()
15         let $f$ be the function that $p$ belongs to
16         $P$.annotate($f$)
17         if $p$ define by allocation instruction then
18             continue
19         for each $e \in G$.InEdge($p$) do
20             if $e.source$ is pointer then
21                 $W$.append($e.source$)
22     return $P$
```

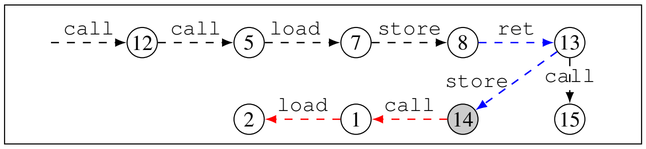
and returned pointers. Then, for each pointer extracted from W, it annotates the functions where the pointer belongs to. Subsequently, if p is defined by an allocation instruction, it indicates that the memory allocation point has been found and no further processing is required; otherwise, all pointers from incoming value-flow edges to current pointer are appended into W, and the process iterates until W becomes empty.

Forward Traversal. The forward traversal constitutes the inverse process of backward traversal, tracing all paths leading to usage points of multilingual objects along value-flow edges. We omit its algorithm description because the overall process is similar to backward traversal, with only two minor differences: First, in contrast to the backward traversal, it appends all pointers connected via outgoing value-flow edges from the current pointer into worklist W. Second, since forward traversal exclusively encounters pointer use sites within W, it is unnecessary to determine whether p is defined by allocation instruction.

The program slicing algorithm successively performs forward and backward traversal to generate the annotated program. It also computes all entry functions within the annotated program that do not have any annotated callers, as illustrated in line 5. These entry functions will serve as the starting points for subsequent program analysis.

```
1  define ptr @CMS_sign(ptr %bio) {
2    %d = load %bio ; use-after-free
3  }
4
5  define ptr @Bio::new(ptr %0) {
6    %p = call alloc() ; Object ID: 1
7    %1 = load %0
8    store %p, %1
9    ret %p
10 }
11
12 define void @sign(ptr %data) {
13   %1 = @Bio::new(%data)
14   store %bio, %1
15   dealloc(%1)
16   %cms = CMS_sign(%bio)
17 }
```

(a) The simplified LLVM IR.



(b) The value-flow graph. The number in circle represent the line number of the statements or the formal parameters in Fig. 3a.

Fig. 3: A running example of the program in Fig. 1.

To better illustrate the program slicing, we present a running example using the program shown in Fig. 1. The simplified version of its IR and VFG is shown in Fig. 3. The algorithm starts the pointer parameter `%bio` (marked with gray background) to perform the backward traversal and forward traversal that are represented in blue and red arrows in figure. The backward traversal terminates at line 8 because pointer `%p` is defined by an allocation instruction. Ultimately, all three functions are annotated, and since the traversal does not reach any callers of function `sign`, so it also be considered as a entry function.

IV. Static Analysis and Bug Detection

In this section, we introduce how our static analysis is designed to compute the allocation and deallocation state of the memory objects on the annotated program along with the bug detection methodology based on the analysis results.

A. Points-to Analysis

The analysis target of Rusty is the memory objects which is stored in the pointer variables. Therefore, to identify the points-to relationships between pointers and memory objects, we first perform a points-to analysis prior to abstract interpretation. Rusty employs a context-sensitive, flow-sensitive and demand-driven points-to analysis [27] [28], building upon the state-of-the-art SVF framework [24]. We utilize demand-driven analysis because we only requires points-to results within specific context during abstract interpretation, eliminating the need from whole program analysis. We have implemented several modifications to SVF to improve its compatibility with Rust and our static analysis methodology. Specifically, we integrated Rust-specific allocation functions, including `__rust_alloc` and `__rust_alloc_zero`, to properly aware allocation sites of Rust objects. Furthermore, we introduced some customized override rules for certain Rust functions to override their original implementation to enable more precise pointer alias recognition. After points-to analysis, each pointer variable v is associated with a points-to set

$$ptr(v) = \{id_1, id_2, ..., id_n\}, \tag{1}$$

containing memory objects represented by their object identifiers id.

B. Abstract Interpretation

Based on the points-to analysis results, we next perform abstract interpretation [25] [26] [46] [47] to compute the allocation and deallocation state for each memory object in annotated program. We employ abstract interpretation approach for our analysis because it approximates the semantics of programs without execution and provides high code coverage, and compared to other methods like symbolic analysis, can effectively reduce computational overhead. Abstract interpretation models the program execution based on dedicated abstract domain and a set of transfer functions. The abstract domain represents the abstract state of the program elements, for instance, this represents the memory allocation and deallocation state in our analysis. The transfer functions are defined for each statement to model them how to manipulate the abstract states. Abstract interpretation begins by assigning an initial abstract state to each program element, then simulates execution by applying the transfer functions to update and propagate the abstract state throughout the program. Both the abstract domain and transfer functions should be designed according to the specific analysis requirements. Next, we formally define these components and then describe the execution process of our abstract interpretation using a fixed-point algorithm.

Abstract Domain. RUSTY's static analysis is designed to detect the allocation and deallocation state of memory objects. To formally model these states, our design adopts the monotone framework theory [48] and characterizes the memory object's state by two complete lattice L_{Alloc} and L_{Free} with the finite height, representing the abstract states of allocation and deallocation, respectively. Both of two lattices are defined by their partial order relations $\sqsubseteq$ along with the *join* operation $\sqcup$. Specifically, the lattice L_{Alloc} forms a partial order $(S, \sqsubseteq)$ where the set S contains the allocation state including $RAlloc$, $CAlloc$, indicating whether the memory object is allocated in Rust heap or C/C++ heap. To represent the default and uncertain states, we also define two special state $\bot \in L_{Alloc}$ denoting the uninitialized state, and $\top \in L_{Alloc}$ denoting the all possible state. The binary relation $\sqsubseteq$ specifies the ordering of the state, for example $\bot \sqsubseteq RAlloc$ and $RAlloc \sqsubseteq \top$. Besides, the join operation $\sqcup$ determines how to merge two state of a same object, such as when encountering the control flow merges, for example $RAlloc \sqcup CAlloc = \top$. The lattice L_{Free} of deallocation state follows an analogous definition to L_{Alloc}, containing two deallocation states $RFree$ and $CFree$ along with the special state and corresponding ordering relation and join operation. Moreover, To comprehensively characterize memory object states, we define the product lattice $\mathbf{AS} = L_{Alloc} \times L_{Free}$, constructed as the Cartesian product of the two component lattices

$$\mathbf{AS} : L_{Alloc} \times L_{Free} = \{(x, y) | x \in L_{Alloc}, y \in L_{Free}\} \tag{2}$$

Furthermore, each program point typically maintains multiple active memory objects during execution. To formally describe the execution state at a given program point, we introduce a map lattice $\mathbf{M} : id \rightarrow \mathbf{AS}$ to represent a set of mapping form the object identifiers id to their corresponding abstract state $\mathbf{AS}$. Intuitively, a map lattice is a lookup table that uses id as key and map it to its abstract state. Finally, our abstract domain is also defined another map lattice $\mathbf{AD} : \mathbf{B} \rightarrow \mathbf{M}$, where $\mathbf{B}$ denotes the set of problem points that immediately after the basic block in the control-flow graph (CFG).

Transfer Function. Based on the abstract domain, we present the transfer functions we are defined that model how program statements update and propagate the abstract states. Each transfer function takes the abstract states at the program point immediately before the statement (denoted as In) as input and produces the abstract states at the point immediately after the statement (denoted as Out), as formalized by the equation

$$Out[s] = \mathcal{V}(s, In[s]), \tag{3}$$

where s represents a statement and $\mathcal{V}$ denotes the transfer function. When analyzing a statement, the transfer function first retrieves the abstract state of its operands from the In set if needed, then updates the operand states in the Out set based on the statement semantics. As described in our abstract domain implementation, our analysis primarily tracks the allocation and deallocation state of memory objects. Therefore, our transfer functions concentrate on instructions that invoke allocation and deallocation functions, defined as follows:

$$\mathcal{V}(p = alloc_{\mathtt{r}}(), \sigma) = \sigma[id \mapsto RAlloc] \tag{4}$$

$$\mathcal{V}(dealloc_{\mathtt{r}}(p), \sigma) = \sigma[id \mapsto RFree \sqcup \sigma(id)] \tag{5}$$

$$\mathcal{V}(p = alloc_{\mathtt{c}}(), \sigma) = \sigma[id \mapsto CAlloc] \tag{6}$$

$$\mathcal{V}(dealloc_{\mathtt{c}}(p), \sigma) = \sigma[id \mapsto CFree \sqcup \sigma(id)] \tag{7}$$

where $id \in ptr(p)$, the subscript `r` and `c` denote Rust and C/C++, respectively. For allocation function, since this is the only definition of p in the static single assignment form of LLVM IR, the transfer function only needs to update the state of objects pointed to by p to either $RAlloc$ or $CAlloc$. Regarding deallocation function, the transfer function first queries the abstract state from the input σ, then joins it with $RFree$ or $CFree$ to update the state of the objects pointed to by p.

Furthermore, for calls to annotated functions that are neither allocation nor deallocation functions, our transfer function initiates an interprocedural analysis, and then merges the results into the Out set, as formalized as follows:

$$\mathcal{V}(p_1 = f(p_2), \sigma) = \sigma[id \mapsto \sigma(id) \sqcup Out[f](id)] \tag{8}$$

Algorithm 2: Fixed-point Algorithm of RUSTY.

```
Input: $F$: The function represented by CFG
Output: $Out$: The $Out$ set of each basic blocks
Function FixedPointAlgorithm($F$):
    $W \leftarrow \{\texttt{EntryBasicBlock}\}$
    for each basic block $n \in F$ do
        $Out[n] \leftarrow \bot$
    while $W \neq \emptyset$ do
        $n \leftarrow W$.removeNext()
        $In[n] \leftarrow \bigsqcup_{p \in pred[n]} Out[p]$
        for each instruction $i \in n$ do
            $In[n] \leftarrow \mathcal{V}(i, In[n])$
        if $In[n] \neq Out[n]$ then
            $Out[n] \leftarrow In[n]$
            for each successor $s \in succ[n]$ do
                $W$.append($s$)
    return $Out$;
```

where $id \in ptr(p_1) \cup ptr(p_2)$ because the states of the objects pointed to by p_1 and p_2 can both be modified by the function f, and $Out[f]$ denotes the abstract state at the return point of the callee function.

C. Fixed-point Algorithm

To compute the abstract states, we design a fixed-point algorithm [49] [29] to perform the abstract execution on the annotated program. As depicted in Algorithm 2, the algorithm traverses the CFG of the program and iteratively applies transfer functions to statements to update the abstract state at each program point, Our fixed-point is implemented as the classic worklist-based approach, iterating at the granularity of basic blocks to avoid maintaining excessive abstract states during iteration. Initially, the worklist W contains only the entry basic block of the function, while the Out set of each basic block is initialized to uninitialized state $\bot$. The algorithm repeatedly selects a basic block n from W, computes its In set by joining the Out set of its predecessor blocks, then traverses each statement in n and applies transfer functions to update the state within In set. If any state change occurs compared to old Out set, all successor basic blocks of n should be appended into the worklist for re-analysis. Since the lattices we defined have finite height and all transfer functions are monotonic, the algorithm is guaranteed to converge to a fixed point and terminate. RUSTY applies this fixed-point algorithm to each entry function until abstract states are computed for the entire annotated program.

Moreover, the transfer function $\mathcal{V}$ initiates an interprocedural analysis when encountering a function call to an annotated function. Our interprocedural analysis employs the call string based approach with the maximum length of call string of three. We choose call string based approach over alternatives such as the functional approach because the latter requires maintaining excessive function contexts, which may introduce unnecessary analysis overhead to our analysis. Specifically, it would require maintaining the abstract states of all memory objects in $ptr(p)$ where the p is the function pointer parameters, with any state changes triggering a new analysis iteration. In our observations, this approach does not significantly improve precision while incurring substantial computational overhead in our analysis.

As an example, RUSTY performs the fixed-point algorithm for the program in Fig. 3 on the entry function `sign`. After the computation, the abstract state of line 2 is as follows:

$$In[2] = [1 \mapsto (RAlloc, RFree)] \tag{9}$$

TABLE I: Rules for bug detection in RUSTY.

Bug Type[1]	**Prog. Point**[2]	**Target Obj**	**Abstract State** $(L_{Alloc}, L_{Free}) = ?$
UB	OF	All	$(RAlloc, CFree) \vee (CAlloc, RFree)$
LEAK	OR	Owned	$(RAlloc, \bot) \vee (CAlloc, \bot)$
UAF	IUI	All	$(\_, CFree) \vee (\_, RFree)$
DF	IDI	All	$(\_, CFree) \vee (\_, RFree)$
NPD	IUI	All	$(\bot, \bot)$

[1] The bug types includes undefined behavior (UB), use after free (UAF), double free (DF), memory leak (LEAK) and null pointer dereference (NPD).
[2] The program point includes the *out* point of FFI callsite (OF), the *out* point of return instruction (OR), the *in* point of *use* instruction (IUI) and the *in* point of deallocation instruction (IDI).

D. Bug Detection

Bug detection leverages the static analysis results to identify potential memory bugs by examining the state at critical program points. We present detailed detection rule as shown in Table I. The bug detection traverses the CFG and examine the state of the target object according to the rules. For example, we can inspect object's abstract state at the In point of line 2 in Fig. 3 based on the result of equation (9). The abstract state of the object 1 is $(RAlloc, RFree)$, implying that it has already deallocated in Rust, which indicate that the access to it will incur a use-after-free bug.

To achieve higher detection precision of memory leak, we also introduce an analysis to find the memory objects whose ownership belong to the current function, called *owned objects*, based on the ownership information we are obtained from the information and IR generation step. These includes objects whose ownership was moved into current function via parameters or those allocated within the current function but not moved to other functions. Intuitively, the owned objects should be deallocated after the function returned as they will not be used elsewhere, otherwise it will incur a memory leak bug.

When a bug is detected, RUSTY generates diagnostic messages and uses debugging information to map the bug's location from IR back to the source code. Developers can then manually inspect source code to confirm and fix the bugs.

V. IMPLEMENTATION

We implement a RUSTY prototype in Rust and C++ (in 3,000+ LOC) using rustc 1.73.0-nightly and LLVM 16. The

TABLE II: Experimental results on the micro-benchmark.

Case	Bug Type	# of Bug	IR LOC	Time(s) / per line(ms)	`rustc`	MirChecker	Rudra	FFIChecker	RUSTY (Our work)
1	UB/UAF/DF	3	347	0.35 / 1.01	✘	✘	✘	1	✔
2	DF	1	358	0.38 / 1.06	✘	✘	✘	✘	✔
3	UB/DF	2	332	0.33 / 0.99	✘	✘	✘	1	✔
4	UAF/DF	2	356	0.31 / 0.87	✘	✘	✘	✘	✔
5	LEAK	2	283	0.31 / 1.10	✘	✘	✘	1	✔
6	LEAK	1	41	0.30 / 7.32	✘	✘	✘	✘	✔
7	NPD	1	105	0.31 / 2.95	✘	✘	✘	✘	✔
8	UAF/DF	2	121	0.29 / 2.40	✘	✘	✘	✘	✔
9	UAF	1	345	0.30 / 0.87	✘	✘	✘	✘	✔
10	UB/LEAK	4	286	0.38 / 1.33	✘	✘	✘	1	✔
11	UB	1	110	0.32 / 2.91	✘	✘	✘	✘	✔
12	UB/LEAK	2	282	0.36 / 1.28	✘	✘	✘	1	✔
Total		22	2,966	3.94 / 1.33	0	0	0	5	22

prototype consists of two primary components: 1) a tool implementing our analysis algorithm that operates directly on LLVM IR files, and 2) a sub-command for Rust's official cargo build system, which can be invoked like standard cargo sub-commands to generate information and IR. The sub-command achieves information generation by implementing a customized `rustc` driver that inserts a callback function into the rustc compilation pipeline to analyze HIR. For LLVM IR generation from Rust code, we inject an additional `emit` flag into rustc prior to compilation. For C/C++ programs, we capture clang commands via custom compilation scripts to extract their corresponding LLVM IR. Our points-to analysis builds upon the SVF framework [24]. To adapt this framework for our purposes, we extend it to recognize Rust allocation functions by integrating their respective function signatures.

VI. EVALUATION

To understand the effectiveness and performance of RUSTY, we evaluate it on micro-benchmarks and real-world Rust programs. Specifically, our evaluation aims to answer the following research questions:

RQ1: Effectiveness. Given that RUSTY is designed to detect multilingual memory bugs for Rust, is RUSTY effective in achieving this goal?

RQ2: Performance. How about the performance of RUSTY in bug detection? Can RUSTY detect multilingual memory bugs within a reasonable time consumption?

RQ3: Ablation Study. As a design objective, does the design approach RUSTY has adopted enhance the effectiveness and performance of multilingual memory bug detection?

RQ4: Comparison Study. Does RUSTY outperform existing static detectors of multilingual memory bugs?

All experiments and measurements are performed on a server with one 20 physical Intel i7 core CPU and 128 GB of RAM running Ubuntu 22.04.

A. Datasets

We conduct the evaluation using two datasets: 1) a set of micro-benchmarks, consisting of 12 vulnerable programs we built; and 2) a real-world CWEs.

Micro-benchmarks. We manually built a micro-benchmark consisting of 12 benchmarks and total 22 bugs covering five different multilingual memory bug types, as shown in Table II, including undefined behavior (UB), double-free (DF), use-after-free (UAF), and so on. These benchmarks are collected from common Rust multilingual memory bug patterns on RustSec [50] and existing literature on Rust security studies. To better reflect the essence of these bugs and to simplify the validation, we have rewritten some of the original buggy code by removing irrelevant code.

Real-world CWEs. To evaluate the effectiveness of RUSTY in ubiquitous applications, we leverage CWE [30] [31] as our real-world benchmarks, and select 50 C/C++ test cases from the common weaknesses in written in C and C++. Our selection is guided by two principles: First, since our focus on detecting Rust multilingual memory bugs, we only chose the types of memory weaknesses. Second, the CWE must be successfully compilable without undefined function or incomplete data structures, and without requiring specific support from specific tools. To use CWE for the evaluation of RUSTY, we added a Rust wrapper to each case in CWE, turning them into multilingual Rust applications.

B. RQ1: Effectiveness

To answer RQ1, we first apply RUSTY to the micro-benchmarks to assess its effectiveness. The experimental results are presented in the last column of Table II, demonstrating that RUSTY can successfully detect all memory bugs in these benchmarks. Furthermore, we further verify the accuracy of the warning locations and bug types, and we found that RUSTY can also effectively identify the correct bug types and pinpoint the exact source lines. These results demonstrate that RUSTY can effectively detect multilingual memory bugs and provide precise bug types and locations.

To investigate the effectiveness of RUSTY on real-world programs, we applied RUSTY to the CWEs. As shown in the first column of Table III, 38 bugs are successfully detected by RUSTY whereas 12 are not. Hence, the true positive is 76% ($Found/Total\ cases$). To further investigate the root causes for the above 12 failed test cases, we conducted a manual

TABLE III: Experimental results on the real-world CWEs

Metric	**RUSTY**	**RUSTY^{wp}**	**RUSTY^{nc}**	**FFIChecker**
Bug found	38	40	23	22
Bug miss	12	10	27	28
True positive	76%	80%	46%	44%
Time(s)	1.24	16.62	1.03	2.21
Memory(MB)	91	956	86	179

inspection of their IR. Our analysis revealed two important reasons: 1) undefined external functions; and 2) conflicting conditions. Rust provides some bindings for external function interfaces by default, but the definitions of these functions are not included in the generated IR, so RUSTY cannot analyze those functions. For example, due to the lack of the function body of `CString::new`, RUSTY cannot recognize that it will allocate a memory object and then analyze its state. Although establishing precise models for those functions will lead to more accurate analysis, this task is tedious and laborious. Another reason is the conflicting conditions, stemming from that static analysis cannot properly handle conditions and branches. This may lead to the situations where some opposing code blocks are considered executable simultaneously, leading to an impossible abstract state for some objects. But this is an inherent limitation of static analysis and should not be deemed as a limitation of RUSTY.

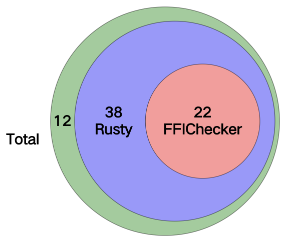
C. RQ2: Performance

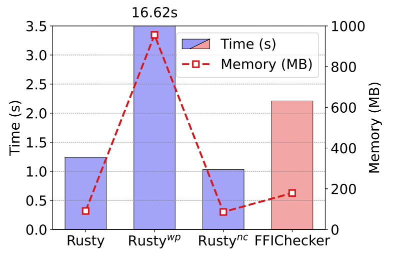
To answer RQ2, we investigate its practical performance, we first apply RUSTY to micro-benchmarks and count the running time of each test case, as shown in the 5th column in Table II. We run each test case five times, then calculate the average time. The experimental results show that the average time spent on micro-benchmarks is around 0.33 seconds, with 1.33 milliseconds per line of code. Furthermore, We apply RUSTY to real-work CWEs and count their running times and peak memories. As shown in the first column of Table III, the results show that each test case take an average of 1.24 seconds, with the average peak memory of 91 MB. These results demonstrate that RUSTY is efficient to detect multilingual memory bugs in Rust applications.

D. RQ3: Ablation Study

To answer RQ3, we investigate whether the program slicing and context-sensitive we introduced can improve the performance and efficiency of bug detection. We first disable the program slicing and context-sensitive and constructed two variants of RUSTY, called RUSTY^{wp} and RUSTY^{nc}, respectively. Since RUSTY^{wp} no longer focuses on the annotated functions, its detection starts from all public functions and the main function.

We evaluate two variants on real-world benchmarks and run each test case 5 rounds then calculate their average execution time and peak memory consumption, as shown in the second and third row of Table III. RUSTY^{wp} takes an average of 16.62 seconds and 956 MB memory to detect each test case, which is 15.38 seconds longer and 865 MB memory more





(a) Bug detection capability. (b) Evaluation metrics.

Fig. 4: Comparison of experimental results between RUSTY and FFIChecker on real-world CWEs.

than RUSTY. However, with so much resource consumption, RUSTY^{wp} only detects two more bugs than RUSTY. We argue that the resource consumption incurred to detect few extra bugs is not worthwhile, and this is particularly crucial for large-scale projects. Besides, without the context-sensitive, RUSTY^{nc} detected only 23 bugs. We further examine the analysis process of the failed test cases and identified two main reasons: First, context-insensitive points-to analysis may cause a pointer points to more objects, even though some of these objects do not actually belong to its real points-to set, leading to incorrect updates of the state of some memory objects. In addition, context-insensitive abstract interpretation cannot distinguish between different invocations of the same function, so a function call statement should join all possible interprocedural analysis results of the callee function, leading to imprecision. In summary, our experimental results demonstrate that the program slicing and context-sensitive in our approach can effectively improve the efficiency and accuracy of detection.

E. RQ4: Comparison Study

To answer RQ4 and understand RUSTY's technical advantages, we compare RUSTY with state-of-the-art static analysis tools related to our work, including MIRChecker [41], Rudra [42] and FFIChecker [18]. We first evaluate the detection ability of them on micro-benchmarks. The result is shown in the 7th to 9th columns of Table II. The first two analysis tools cannot detect any bugs caused by multilingual interactions, because these tools are based on Rust IRs and treat external C/C++ code as a black box. Besides that, FFIChecker only finds 5 bugs in total due to the limitations of its analysis which only consider limited multilingual interaction patterns.

Furthermore, we compare the detection ability and performance of RUSTY and FFIChecker on our real-world CWEs. As shown in the last column of Table III and Fig. 4b, FFIChecker finds 22 bugs with a true positive rate of 44% and average time and memory consumption of 2.21 seconds and 179 MB. More importantly, as shown in the Venn diagram 4a, RUSTY is capable of detecting all the bugs that FFIChecker can identify, which demonstrates that RUSTY fully covers the detection capabilities of FFIChecker. In summary, the comparison results indicate that in terms of bug detection ability and efficiency, our approach is outperform existing static analysis tools.

VII. Discussion and Limitation

In this section, we discuss some of our limitations and possible enhancements to this work, along with directions for future work.

Source Code Availability. Our work assumes that the source code of external C/C++ libraries is accessible. However, this may not hold in certain scenarios. For example, external C/C++ libraries might only provide binary-form libraries for reasons such as ease of distribution or copyright issues. We are currently unable to handle such cases. But thanks to the open ecosystem of Rust community, most of the source code is accessible. In the future, we may incorporate dynamic analysis to handle external C/C++ binary libraries, similar to what [11] [20] does. When encountering FFIs without source code, pointer analysis and abstract interpretation can be performed through dynamic execution.

Other Programming Languages. Although RUSTY can effectively detect various multilingual memory bugs as shown by our experimental results, our evaluation was limited to Rust and C/C++ programs. We have not evaluated bugs caused by interactions between Rust and other languages, such as with Fortran, Python, Java, etc. However, in theory, our approach can be easily extended to detect bugs in other combinations with Rust (possibly requiring minor specific modifications) provided those languages can be translated into LLVM IR. Currently, beyond its official subprojects, LLVM incorporates a broad variety of other projects [51] that are capable of compiling these languages into LLVM IR, thereby paving the way for our tool to support additional language pairs.

Future Work. Even though RUSTY effectively detects various multilingual memory bugs, as demonstrated by our experimental results, it may miss some vulnerabilities and still has opportunities for enhancement. For example, our analysis may miss the objects created through forced address casts, such as by converting integer values to pointers. Besides, our analysis does not account for address arithmetic, which may lead to missed out-of-bounds access errors. In this direction, we can collaborate with the latest novel dynamic detection techniques [23] [52] and the isolation techniques [22] [53] [54] to detect or isolate bugs that RUSTY may miss, further enhancing its capabilities. Note that this work represents the first step towards a holistic static analysis for multilingual Rust programs, we leave these as important future work.

VIII. Related Work

In recent years, there has been substantial research on Rust memory bug detection and multilingual bug detection.

Rust Static Analysis. Prior studies [55] [56] [41] [42] [57] have presented many novel static analyzers and found a wide variety of bugs. CuiMohan et al. [55] leverage a static path-sensitive data-flow analysis approach to detect use-after-free and double-free issues. Hua et al. [56] use a lightweight data-flow analysis to detect and automatically rectify buffer overflow vulnerabilities. Li et al. [41] perform a static analysis on Rust's MIR to track both numerical and symbolic information, detects potential runtime crashes and memory-safety errors. Bae et al. [42] identified three bug patterns in unsafe Rust and implemented a static analyzer that can quickly recognize error-prone parts of unsafe code.

However, a significant limitation of these studies is that they are few beyond Rust and account the bugs arisen when Rust interacts with other languages. Our work takes a new step and fills this gap.

Multilingual Bug Detection. There have been a lot of works on multilingual bug detection, including combinations of different languages with C such as Python [20] [19], Java [21], and as well as Rust [17] [18] [11]. To the best of our knowledge, FFIChecker [18] is a pioneering work in Rust multilingual bug detection, which also uses abstract interpretation and leverages taint analysis to analyze program variables. But it solely focuses on Rust's objects and only analyzes other C/C++ if necessary. Hu et al. [17] translated Rust and C programs into customized IR and ported several existing Rust analysis tools, including Miri and MIRChecker, to analyze multilingual programs. McCormack et al. [11] conducted an empirical study on the undefined behavior in multilingual Rust programs using Miri and an LLVM interpreter to jointly execute programs. Mergendahl et al. [49] systematically analyzed the security of multilingual applications and constructed threat models across Rust and C. Moreover, for other languages, Li et al. [19] transform Python/C interface code into affine program to find Python objects' reference-counting errors. Li et al. [20] present a dynamic information flow analysis (DIFA) technique PolyCruise for multilingual software. Lee et al. [21] perform a whole-program analysis with extracted semantic summaries from Java and C programs.

Different from these studies, our work performs a holistic static analysis on Rust and C/C++ and applies a different abstract interpretation approach to address their limitations.

IX. Conclusion

In this work, we present an approach for detecting Rust multilingual memory bugs through static analysis. Our approach leverages abstract interpretation to analyze the memory object states based on the abstract domain and transfer functions we defined for our detection scenarios. We first designed a program slicing algorithm to refine the program and improve the efficiency of program analysis. Subsequently, we utilized points-to analysis and abstract interpretation to analyze the abstract states of memory objects and designed a series of detection rules tailored to different memory bugs. We implement a prototype for RUSTY and conduct experiments to evaluate its effectiveness and performance. The experimental results demonstrate that RUSTY can effectively and efficiently detect Rust memory safety bugs caused by multilingual interactions, outperforming state-of-the-art studies. Overall, our work represents a new step in detecting memory bugs in Rust, further enhancing the guarantees of Rust as a safe language.

References

[1] H. Xu, Z. Chen, M. Sun, Y. Zhou, and M. R. Lyu, "Memory-safety challenge considered solved? an in-depth study with all rust cves," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 1, pp. 3:1–3:25, Sep. 2021.

[2] J. V. Stoep, "Memory safe languages in android 13," Dec. 2022.

[3] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, S. Leonard, P. Pannuto, P. Dutta, and P. Levis, "The tock embedded operating system," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, ser. SenSys '17. New York, NY, USA: Association for Computing Machinery, Nov. 2017, pp. 1–2.

[4] "Rust for linux," https://rust-for-linux.com.

[5] K. Boos, N. Liyanage, R. Ijaz, and L. Zhong, "Theseus: An experiment in operating system structure and state management," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 1–19.

[6] V. Narayanan, T. Huang, D. Detweiler, D. Appel, Z. Li, G. Zellweger, and A. Burtsev, "{RedLeaf}: Isolation and communication in a safe operating system," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 21–39.

[7] "Tikv/tikv: Distributed transactional key-value database, originally created to complement tidb," https://github.com/tikv/tikv.

[8] B. Anderson, L. Bergstrom, M. Goregaokar, J. Matthews, K. McAllister, J. Moffitt, and S. Sapin, "Engineering the servo web browser engine using rust," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, May 2016, pp. 81–89.

[9] "Wasmtime," https://wasmtime.dev/, Jun. 2025.

[10] H. Li, L. Guo, Y. Yang, S. Wang, and M. Xu, "An empirical study of {Rust-for-Linux}: The success, dissatisfaction, and compromise," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, 2024, pp. 425–443.

[11] I. McCormack, J. Sunshine, and J. Aldrich, "A study of undefined behavior across foreign function boundaries in rust libraries," Apr. 2025.

[12] "Mozilla-firefox/firefox: The official repository of mozilla's firefox web browser." https://github.com/mozilla-firefox/firefox.

[13] M. Cui, S. Sun, H. Xu, and Y. Zhou, "Is unsafe an achilles' heel? a comprehensive study of safety requirements in unsafe rust programming," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. Lisbon Portugal: ACM, Apr. 2024, pp. 1–13.

[14] A. Maiga, C. Artho, F. Gilcher, and Y. Moy, "Does rust spark joy? safe bindings from rust to spark, applied to the bbqueue library," in *Proceedings of the 9th ACM SIGPLAN International Workshop on Formal Techniques for Safety-Critical Systems*, ser. FTSCS 2023. New York, NY, USA: Association for Computing Machinery, Oct. 2023, pp. 37–47.

[15] A. N. Evans, B. Campbell, and M. L. Soffa, "Is rust used safely by software developers?" in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 246–257.

[16] Z. Yu, L. Song, and Y. Zhang, "Fearless concurrency? understanding concurrent programming safety in real-world rust software," Feb. 2019.

[17] S. Hu, B. Hua, L. Xia, and Y. Wang, "Crust: Towards a unified cross-language program analysis framework for rust," in *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, Dec. 2022, pp. 970–981.

[18] Z. Li, J. Wang, M. Sun, and J. C. S. Lui, "Detecting cross-language memory management issues in rust," in *Computer Security – ESORICS 2022: 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26–30, 2022, Proceedings, Part III*. Berlin, Heidelberg: Springer-Verlag, Sep. 2022, pp. 680–700.

[19] S. Li and G. Tan, "Finding reference-counting errors in python/c programs with affine analysis," in *ECOOP 2014 – Object-Oriented Programming*, R. Jones, Ed. Berlin, Heidelberg: Springer, 2014, pp. 80–104.

[20] W. Li, J. Ming, X. Luo, and H. Cai, "{PolyCruise}: A {Cross-Language} dynamic information flow analysis," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2513–2530.

[21] S. Lee, H. Lee, and S. Ryu, "Broadening horizons of multilingual static analysis: Semantic summary extraction from c code for jni program analysis," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, Jan. 2021, pp. 127–137.

[22] I. Bang, M. Kayondo, H. Moon, and Y. Paek, "Trust: A compilation framework for in-process isolation to protect safe rust against untrusted code," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 6947–6964.

[23] K. Cho, J. Kim, K. D. Duy, H. Lim, and H. Lee, "Rustsan: Retrofitting addresssanitizer for efficient sanitization of rust."

[24] Y. Sui and J. Xue, "Svf: Interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC '16. New York, NY, USA: Association for Computing Machinery, Mar. 2016, pp. 265–266.

[25] P. Cousot and R. Cousot, "Static determination of dynamic properties of generalized type unions," *SIGSOFT Softw. Eng. Notes*, vol. 2, no. 2, pp. 77–94, Mar. 1977.

[26] ——, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1977, pp. 238–252.

[27] Y. Sui and J. Xue, "On-demand strong update analysis via value-flow refinement," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, Nov. 2016, pp. 460–473.

[28] ——, "Value-flow-based demand-driven pointer analysis for c and c++," *IEEE Transactions on Software Engineering*, vol. 46, no. 8, pp. 812–835, Aug. 2020.

[29] U. P. Khedker, A. Sanyal, and B. Karkare, *Data Flow Analysis: Theory and Practice*, 1st ed. Baton Rouge: Taylor & Francis Group, 2009.

[30] "Cwe - cwe-658: Weaknesses in software written in c (4.17)," https://cwe.mitre.org/data/definitions/658.html.

[31] "Cwe - cwe-659: Weaknesses in software written in c++ (4.17)," https://cwe.mitre.org/data/definitions/659.html.

[32] "Ffi - the rustonomicon," https://doc.rust-lang.org/nomicon/ffi.html.

[33] "Cve-2018-20997," https://www.cve.org/CVERecord?id=CVE-2018-20997.

[34] D. B. Stephens, K. Aldoshan, and M. R. Khandaker, "Understanding the challenges in detecting vulnerabilities of rust applications," in *2024 IEEE Secure Development Conference (SecDev)*. Pittsburgh, PA, USA: IEEE, Oct. 2024, pp. 54–63.

[35] K. R. Fulton, A. Chan, D. Votipka, M. Hicks, and M. L. Mazurek, "Benefits and drawbacks of adopting a secure programming language: Rust as a case study," in *Proceedings of the Seventeenth USENIX Conference on Usable Privacy and Security*, ser. SOUPS'21. USA: USENIX Association, Aug. 2021, pp. 597–616.

[36] S. Höltervennhoff, P. Klostermeyer, N. Wöhler, Y. Acar, and S. Fahl, "{"I} wouldn't want my unsafe code to run my {pacemaker"}: An interview study on the use, comprehension, and perceived risks of unsafe rust," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 2509–2525.

[37] I. McCormack, T. Dougan, S. Estep, H. Hibshi, J. Aldrich, and J. Sunshine, "A mixed-methods study on the implications of unsafe rust for interoperation, encapsulation, and tooling," Oct. 2024.

[38] V. Astrauskas, C. Matheja, F. Poli, P. Müller, and A. J. Summers, "How do programmers use unsafe rust?" *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 136:1–136:27, Nov. 2020.

[39] L. Gäher, M. Sammler, R. Jung, R. Krebbers, and D. Dreyer, "Refinedrust: A type system for high-assurance verification of rust programs," *Proceedings of the ACM on Programming Languages*, vol. 8, no. PLDI, pp. 1115–1139, Jun. 2024.

[40] S. Ho, A. Fromherz, and J. Protzenko, "Sound borrow-checking for rust via symbolic semantics," *Proceedings of the ACM on Programming Languages*, vol. 8, no. ICFP, pp. 426–454, Aug. 2024.

[41] Z. Li, J. Wang, M. Sun, and J. C. Lui, "Mirchecker: Detecting bugs in rust programs via static analysis," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. Virtual Event Republic of Korea: ACM, Nov. 2021, pp. 2183–2196.

[42] Y. Bae, Y. Kim, A. Askar, J. Lim, and T. Kim, "Rudra: Finding memory safety bugs in rust at the ecosystem scale," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. Virtual Event Germany: ACM, Oct. 2021, pp. 84–99.

[43] "Llvm's analysis and transform passes — llvm 20.1.0 documentation," https://releases.llvm.org/20.1.0/docs/Passes.html#mem2reg-promote-memory-to-register.

[44] B. Steffen, J. Knoop, and O. Rüthing, "The value flow graph: A program representation for optimal program transformations," in *ESOP '90*, N. Jones, Ed. Berlin, Heidelberg: Springer, 1990, pp. 389–405.

[45] R. Bodík and S. Anik, "Path-sensitive value-flow analysis," in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '98. New York, NY, USA: Association for Computing Machinery, Jan. 1998, pp. 237–251.

[46] P. Cousot and R. Cousot, "Systematic design of program analysis frameworks," in *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '79. New York, NY, USA: Association for Computing Machinery, Jan. 1979, pp. 269–282.

[47] PATRICK. COUSOT and RADHIA. COUSOT, "Abstract interpretation frameworks," *Journal of Logic and Computation*, vol. 2, no. 4, pp. 511–547, Aug. 1992.

[48] K. B and U. D, "Monotone data flow analysis frameworks," *Acta Informatica*, Sep. 1977.

[49] A. Møller and M. I. Schwartzbach, *Static Program Analysis*.

[50] "Advisories > rustsec advisory database," https://rustsec.org/advisories/.

[51] "The llvm compiler infrastructure project," https://llvm.org/ProjectsWithLLVM/.

[52] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "{AddressSanitizer}: A fast address sanity checker," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 309–318.

[53] L. Schuermann, J. Toubes, T. Potyondy, P. Pannuto, M. Milano, and A. Levy, "Building bridges: Safe interactions with foreign languages through omniglot," in *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, 2025, pp. 595–613.

[54] P. Liu, G. Zhao, and J. Huang, "Securing unsafe rust programs with xrust," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. Seoul South Korea: ACM, Jun. 2020, pp. 234–245.

[55] CuiMohan, ChenChengjun, XuHui, and ZhouYangfan, "Safedrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis," *ACM Transactions on Software Engineering and Methodology*, May 2023.

[56] B. Hua, W. Ouyang, C. Jiang, Q. Fan, and Z. Pan, "Rupair: Towards automatic buffer overflow detection and rectification for rust," in *Annual Computer Security Applications Conference*. Virtual Event USA: ACM, Dec. 2021, pp. 812–823.

[57] B. Qin, Y. Chen, H. Liu, H. Zhang, Q. Wen, L. Song, and Y. Zhang, "Understanding and detecting real-world safety issues in rust," *IEEE Trans. Softw. Eng.*, vol. 50, no. 6, pp. 1306–1324, Jun. 2024.