FFIPROBE: Effectively Detecting Multilingual Rust Memory Bugs with FFI Sanitization

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 promising systems programming language for security-critical domains, offering effective protection against memory safety issues through its strong type system and ownership model. However, practical multilingual Rust applications that interact with unsafe languages such as C/C++ via the Foreign Function Interface (FFI) can undermine Rust's security guarantees because the integration of disparate memory management mechanisms often circumvents Rust's compiler safety checks, rendering the interaction boundaries highly error-prone.**

In this paper, we propose FFIPROBE, a technique that enables dynamic analysis for effectively detecting memory safety bugs in multilingual Rust programs. Specifically, we utilize Rust's mid-level intermediate representation (MIR) as the instrumentation target, statically instrument probe functions for raw pointer operations at multilingual interaction boundaries, and identify memory safety bugs arising from memory objects in illegal states during program execution. To facilitate accurate tracking of memory object states, we also design a custom heap allocator that proxies memory allocation requests from both Rust and foreign languages, while maintaining metadata for allocated memory objects to support probe functions in identifying bugs at runtime. We implement a prototype of FFIPROBE and conduct extensive experiments to evaluate its practical effectiveness and performance on 2,635 real-world multilingual Rust packages. Our evaluation detects 43 memory safety bugs across 29 packages, while introducing an average runtime overhead of 25.6% and a memory overhead of 33.2%.

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

I. INTRODUCTION

Rust has emerged as a highly promising systems programming language for security-critical applications, owing to its unique ownership model and strong type system, which have proven effective in preventing memory safety issues [1].These advantages position Rust as a compelling alternative to C/C++, leading to its successful adoption in security-critical domains, including operating systems [2], databases [3], web browsers [4], and language runtimes [5].

Unfortunately, Rust's memory safety guarantee is not a panacea and is threatened by the growing prevalence of multilingual programming in Rust [7] [8]. Many real-world Rust applications (*e.g.*, Linux [9] and Firefox [4]) are multilingual to interact with external modules written in memory-unsafe languages (*e.g.*, C/C++) via Foreign Function Interfaces (FFIs). However, while this multilingual paradigm offers significant benefits, including legacy code reuse and toolchain flexibility, it simultaneously introduces new safety challenges. Specifically, vulnerabilities residing in external modules circumvent Rust's safety checks [10], thereby undermining its security guarantees. More critically, even if the modules in different languages are secure, vulnerabilities may still emerge at language boundaries [1] [11], due to language disparities including memory models, memory management mechanisms, and data representation strategies, among others. For instance, Fig. 1 presents a CVE vulnerability stemming from an oversight of Rust ownership mechanism. Therefore, detecting memory bugs in multilingual Rust programs is both critical and urgent.

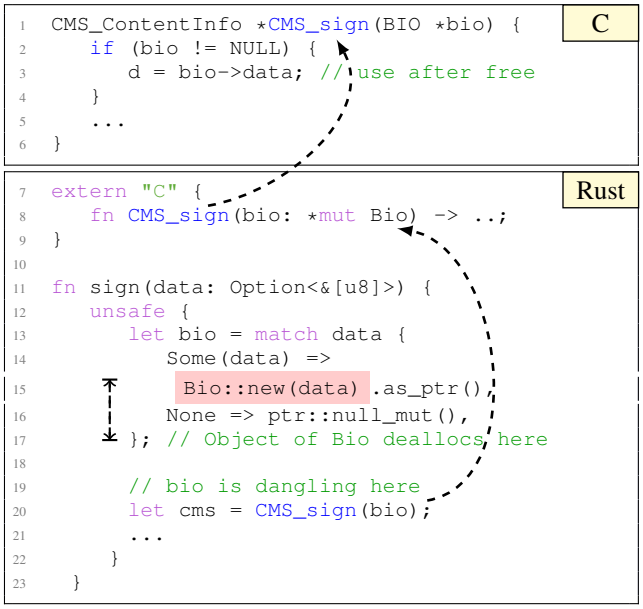


Fig. 1: An example vulnerability that we adapted from CVE-2018-20997 [6], which is uncovered in the `openssl` Rust crate with a critical CVSS score of 9.8 and causes a use-after-free due to improper multilingual interaction.

Recognizing this need, researchers have conducted preliminary studies for analyzing multilingual Rust programs [12] [13] [8] [14]. Unfortunately, despite this promising progress,

* The corresponding author.

there are still largely underexplored areas due to several challenges to be tackled. First, many static analysis methods lack flexibility, as they typically require foreign code to be fully visible and translatable into the intermediate representation (IR) required for analysis. However, this requirement is often difficult to satisfy, as external modules may be provided solely as binary libraries, or the foreign language cannot translate into the necessary IR. The absence of these essential IR renders such analysis infeasible. Second, some approaches are limited to particular scenarios, making it challenging to address more complex multilingual interactions arising in real-world applications. For example, FFIChecker [13] only tracks heap memory allocated in Rust and passed to C/C++, while overlooking heap objects allocated in C/C++ and passed to Rust. Consequently, it struggles to detect memory issues caused by C/C++ objects.

In this paper, to fill this gap, we propose a dynamic analysis approach for detecting memory bugs in multilingual Rust programs. Our approach employs static instrumentation at multilingual interaction boundaries and executes the instrumented detection code at runtime to identify potential multilingual memory safety bugs, which comprises two core components. First, we utilize Rust mid-level intermediate representation (MIR) as the instrumentation target, and develop a MIR pass to instrument probe functions for raw pointer operations at multilingual interaction boundaries during compilation. These probe functions will execute alongside the program to identify memory bugs caused by illegal memory objects and operations at runtime. Second, to facilitate tracking of memory object states, we design a custom heap allocator that proxies memory allocation requests from both Rust and foreign languages while simultaneously maintaining metadata for memory objects, including language affiliation and allocation/deallocation status, to support bug detection by the instrumented probe functions.

To validate our approach, we implement a prototype of FFIPROBE and conduct extensive experiments to evaluate its effectiveness and performance on a dataset of 2,635 real-world Rust multilingual packages. The experimental results demonstrate that, without relying on foreign language source code, FFIPROBE successfully identifies 43 memory safety bugs across 29 packages, while incurring only 25.6% average runtime overhead and 33.2% memory overhead. These results indicate that our approach effectively detects memory safety issues arising from multilingual interactions with favorable resource overhead.

In summary, this paper makes the following contributions:

- We propose a dynamic analysis approach for effectively detecting Rust multilingual memory safety bugs.
- We design and implement a prototype of FFIPROBE to validate our approach.
- We conduct a comprehensive evaluation to demonstrate that our approach is effective and efficient.

Our source code is also publicly available to facilitate further research[1].

[1]https://doi.org/10.5281/zenodo.19626633

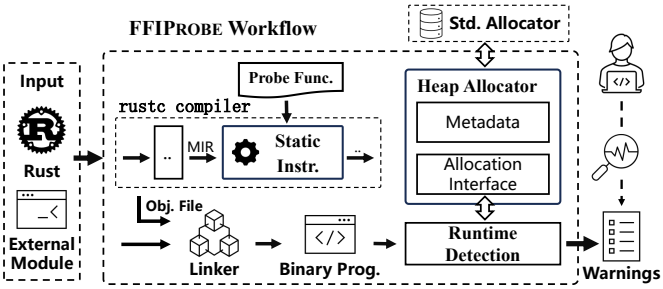


Fig. 2: An overview of FFIPROBE's workflow.

II. APPROACH

In this section, we first provide an overview of our approach (§ II-A), and then introduce its two key components of static instrumentation (§ II-B) and custom heap allocator (§ II-C).

A. Overview

Fig. 2 presents an overview of the workflow of our approach. It accepts Rust source code and external modules as input and outputs bug warnings for the multilingual program. The workflow primarily consists of two phases: static instrumentation and runtime detection, which operate as follows:

In the static instrumentation phase, the Rust compiler accepts Rust source code as input and executes a MIR-based static instrumentation pass at the final stages of the compiler frontend. This pass identifies the boundaries of multilingual interactions and instruments probe functions for statements that involve unsafe raw pointer operations at these boundaries, thereby validating the legitimacy of raw pointers before and after their use. Subsequently, the MIR code undergoes the remaining compilation phases to generate a relocatable object file, which is then linked with external modules and the custom heap allocator to produce an executable binary program.

In the runtime detection phase, the checks of the probe functions are triggered by executing the binary program. During this phase, the custom heap allocator proxies all memory allocation requests within the program, maintaining metadata for memory objects, including language affiliation and allocation/deallocation status. It also provides metadata query interfaces to assist the probe functions in bug detection. Upon completion of program execution, the runtime detection phase generates bug warnings, which developers can then use to review code and fix potential bugs.

B. Static Instrumentation

Instrumentation Sites and Target. We select the boundaries of multilingual interactions as instrumentation sites. Our choice is inspired by numerous studies on Rust safety [15] [16] [17]. Specifically, Rust's ownership mechanism can effectively eliminate memory safety issues. However, once Rust unsafe code is introduced, these safety guarantees may be circumvented [18] [1]. Existing research indicates that if unsafe code in a program (*e.g.*, FFI calls) can be verified to satisfy its preconditions and invariants, the overall safety of the program can still be guaranteed [19]. This implies that our

TABLE I: The probe functions of FFIPROBE, including their instrumentation points and the bugs that arise when sanitization rules are violated.

Prog. Position	Instr. Point	Probe Func.	San. Rule[1]	Bug[2]
FFI call stmt.	Entry	`pre_ffi`	(1)	UAF
	Exit	`post_ffi`	(2)	UB
			(3)	OOB
			(4)	UAF
Stmt. using raw pointers	Entry	`ptr_use`	(1)	UAF
			(5)	NPD
Exported function	Entry Exit	`ptr_nofree`	(1) (4)	UAF
Drop stmt.	Entry	`pre_free`	(1)	DF
			(2)	UB
Program exit stmt.	Entry	`pre_exit`	(6)	LEAK

[1] The sanitization rules include: (1) pointers should not be freed; (2) allocation and deallocation states should be consistent; (3) the contents of red zones should remain intact; (4) returned pointers should not be freed; (5) pointers should not be null; (6) all multilingual objects must be properly freed.

[2] The multilingual memory bugs include use-after-free (UAF), double-free (DF), undefined behavior (UB), null pointer dereference (NPD), out-of-bounds memory access (OOB), and memory leaks (LEAK).

detection does not need to monitor the entire program; instead, attention should be directed toward high-risk areas that breach Rust's safety boundaries. Therefore, our instrumentation focuses on unsafe operations of raw pointers at the boundaries of multilingual interactions, thereby avoiding whole-program instrumentation and minimizing runtime overhead.

Furthermore, we select Rust MIR as the instrumentation target. MIR offers several advantages compared to other alternatives: First, its syntax is concise, facilitating analysis and transformation. Second, it possesses rich semantic information, enabling effective identification of FFI and raw pointer operations. Third, it is flexible and has been thoroughly validated as a viable representation for program analysis [14].

Probe Function. Directly embedding detection logic into the MIR is typically complex and prone to introducing excessive modifications. To address this, we encapsulate the detection code into multiple probe functions, each tailored to specific detection requirements. These functions are subsequently inserted into the MIR as function calls, thereby minimizing intrusion into the MIR and simplifying the instrumentation design. Table I enumerates these probe functions, along with their corresponding sanitization rules and the bug types triggered when these rules are violated.

Instrumentation Algorithm. Based on the design of the aforementioned probe functions, we implemented a static instrumentation pass to insert these functions into the target program, as presented in Algorithm 1. This pass is scheduled at the final stage of the `rustc` frontend compilation pipeline.

MIR enforces a strict structure in which function calls are treated as terminating statements within basic blocks. Therefore, to preserve this structural integrity, inserting probe functions requires not only introducing new function calls but also creating additional basic blocks. Furthermore, this algorithm does not perform instrumentation at `Drop` statements or program exit point. Since the generic monomorphization of `Drop` has not been completed at the stage at which the instrumentation pass operates, we lower its instrumentation to the `dealloc` interface of the Rust global allocator. Regarding memory leak detection, we handle this centrally prior to program exit rather than at interaction boundaries.

Algorithm 1: Static Instrumentation Algorithm

Input: The MIR F of the function at boundaries
Output: The MIR after instrumentation

Function `Instrumentation`(F)**:**
  for *each basic block* $bb \in F$ **do**
    $args \leftarrow$ GetRawPointer$(bb.term)$
    if $args \neq \varnothing$ **then**
      if $bb.term$ *is a FFI call* **then**
        INSERT$(bb.term$, `pre_ffi`, $args)$
        APPEND$(bb$, `post_ffi`, $args)$
      else
        INSERT$(bb.term$, `ptr_use`, $args)$
    for *each statement* $s \in reverse(bb.stmts)$ **do**
      $ptrs \leftarrow$ GetDereferencedPointer(s)
      if $ptrs \neq \varnothing$ **then**
        INSERT$(s$, `ptr_use`, $ptrs)$
  if F *is an exported function* **then**
    if F *has pointer arguments* $ptrs$ **then**
      INSERT$(F.entry$, `ptr_nofree`, $ptrs)$
    if F *returns pointer* ptr **then**
      INSERT$(F.return$, `ptr_nofree`, $ptr)$

Function `Insert`$(cur,\ fn,\ args)$**:**
  $stmts, term \leftarrow cur.bb$.splitOff$(cur)$
  $newbb \leftarrow$ NewBlock$(stmts, term)$
  $cur.bb.term \leftarrow$ NewCall$(fn, args, newbb)$

Function `Append`$(bb,\ fn,\ args)$**:**
  $term \leftarrow$ NewCall$(fn, args, bb.term.dest)$
  $bb.term.dest \leftarrow$ NewBlock$(\varnothing, term)$

C. Custom Heap Allocator

The design of FFIPROBE's custom heap allocator overrides the allocation interfaces of the standard C allocator. Given that the design of a high-performance memory allocator is beyond the scope of this study, the memory allocation interface is implemented as a wrapper around existing standard allocator interfaces, delegating the actual memory allocation and deallocation operations to the standard allocator. To facilitate unified management of object metadata across different languages, we employ a single allocator to provide memory allocation services and metadata management for both Rust and foreign languages via three categories of interfaces: 1) C interfaces,

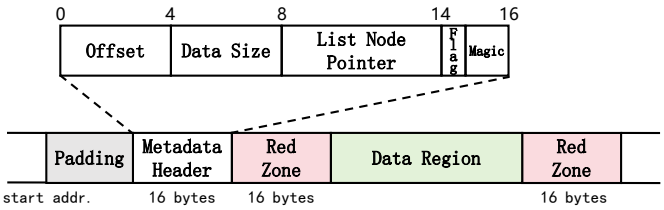


Fig. 3: The memory object layout in custom heap allocator.

2) Rust interfaces, and 3) metadata management interfaces. Besides, we also implement a new Rust global allocator based on the Rust interface to support high-level memory allocation operations in Rust, such as `Box::new` and collection operations.

Memory Object Layout. The heap allocator is required to provide corresponding metadata information or mechanisms to assist in bug detection. To this end, we design a specific memory object layout tailored to multilingual memory bugs, as shown in Fig. 3, which primarily comprises four key parts:

First, the data region is the area returned by the allocator to the upper-level application for storing user data. Its size is determined by the allocation request, and all parts other than this region are transparent to the upper-level application.

Second, the metadata header stores the metadata of the memory object, with a fixed size of 16 bytes. It is placed immediately before the left red zone of the data region, enabling rapid locating from the data region pointer based on the fixed-size left red zone. Within this header, the offset field records the number of bytes from the start of the memory object to the data region, which is used to reverse-calculate the memory object's starting address from the data region pointer. The data size field records the size of the data region, which is used to locate the right red zone based on the data region pointer. The list node pointer stores a pointer to a node of the multilingual object list using 6 bytes. Given the canonical addressing is widely adopted in 64-bit architectures, we omit the storage of the upper 16 bits of pointers, which are all zeros in user space and utilize the saved space to store the flag and magic number. The flag field occupies 4 bits, using 2 bits each to record an object's allocation and deallocation language affiliation, respectively, which are set during object allocation and deallocation. The *magic number* field occupies 12 bits and is used to identify and verify the validity of the metadata header.

Third, we establish red zones on both sides of the data region to assist in detecting out-of-bounds memory accesses. These red zones are filled with specific magic numbers during memory allocation, allowing probe functions to verify its integrity and thereby determine whether an out-of-bounds access has occurred.

Fourth, we insert a padding region before the metadata header to ensure that the data region is aligned to a specified boundary, thereby satisfying the alignment requirements of certain memory allocation interfaces (*e.g.*, `posix_memalign`). Suppose the required alignment for the data region is n bytes, shifting the data region backward by at most $n-1$ bytes will locate the first position satisfying the alignment. Therefore, to guarantee alignment under all circumstances, the heap allocator will request an additional $n-1$ bytes of space from the standard allocator to accommodate the maximum possible padding.

Multilingual Object List. The detection of memory leaks requires tracking allocated but unreleased memory objects throughout the program execution lifetime. Therefore, the allocator must provide a mechanism to record these objects and support subsequent queries. To this end, we maintain a list within the allocator to record relevant memory objects, and only record multilingual memory objects to minimize the memory overhead given our work. A multilingual memory object will be added to this list when it crosses a language boundary and is passed to another language. This task is accomplished by probe functions instrumented before and after FFI call statements.

Deferred Deallocation Queue. Many standard allocators organize free blocks of the same size into a LIFO free list to improve cache locality. However, such rapid reuse of freed memory objects may corrupt their metadata, thereby causing misjudgments. For example, a freed block may be immediately reallocated, resulting in the overwrite of its original metadata before probe functions query.

To address this problem, we introduce a deferred deallocation queue organized as a fixed-size ring buffer. When a memory object is freed, the allocator does not immediately return it to the standard allocator; instead, it temporarily stores the object at the tail of the queue. The actual deallocation is performed only when the queue is full, at which point the earliest enqueued object is popped from the head. This mechanism ensures that the metadata of a memory object remains valid and accessible for a period of time after deallocation.

III. Evaluation

To systematically evaluate FFIProbe, our evaluation aims to answer the following research questions:

RQ1: Effectiveness. Can FFIProbe effectively detect multilingual memory bugs? Does it mitigate the reliance on foreign source code inherent in static analysis approaches?

RQ2: Performance. How is the detection performance of FFIProbe? What is the resource overhead introduced by static instrumentation?

All experiments were performed on an Ubuntu 22.04 server equipped with a 20-core Intel CPU and 128 GB RAM.

A. Dataset

We construct the experimental dataset based on the crates.io package registry. Specifically, we first crawl packages under the "External FFI bindings" category from this registry. Packages in this category serve as bindings for FFIs, which not only bind to external modules written in C/C++ but also bind to binary modules or modules in other languages (*e.g.*, Go and Haskell). Using these packages as dependencies, we further crawl packages that directly depend on them. From these,

TABLE II: Experimental results of FFIPROBE.

#	Package	Version	Bug Type	# of Bugs	Time		Memory		LoC[1]	Code Size (KB)
					Cost (s)	Increase	Cost (MB)	Increase		
1	gotpl	0.2.6	UB	1	1.33	13.7%	294	0.1%	851	3872
2	tcpp	0.1.0	LEAK	2	0.16	33.3%	166	17.1%	154	1693
3	physfs-rs	0.1.6	LEAK	1	0.08	14.3%	100	15.8%	4475	909
4	minimp4	0.1.2	LEAK	1	0.10	25.0%	100	15.4%	5806	3910
5	wabt	0.10.0	LEAK	1	0.60	30.4%	204	39.2%	1491	4005
6	husk	0.1.3	UB	1	0.25	56.2%	174	43.0%	874	1502
7	regex-rs	0.1.0	LEAK	2	0.24	41.2%	174	52.4%	181	1673
8	wfc-rs	0.6.1	UB	1	0.90	1.1%	13	15.1%	234	1670
9	libmodbus-rs	0.8.3	OOB	1	1.11	35.4%	176	48.9%	2776	2738
10	voxelizer	1.0.0	LEAK	5	0.06	0.0%	101	41.8%	265	1563
11	rooster	2.14.0	LEAK	1	24.42	42.5%	126	29.5%	23992	3219
12	tidesdb-rs	0.1.2	UB	1	1.52	0.7%	134	47.2%	1733	1136
13	spglib	1.15.1	UB	9	0.16	14.3%	164	60.8%	1815	863
14	libucl	0.2.3	LEAK	1	0.20	25.0%	165	58.2%	995	2863
15	vampire-prover	0.2.0	LEAK	1	0.53	51.4%	184	13.2%	572	12453
Average					2.11	25.6%	152	33.2%	3081	2938

[1] Due to the complexity of package dependencies, we count only the lines of Rust code within the package itself.

we select packages containing unit tests that invoke FFIs as our dataset. Finally, we construct a dataset comprising 2,635 packages and utilize their unit tests as the running targets.

B. RQ1: Effectiveness

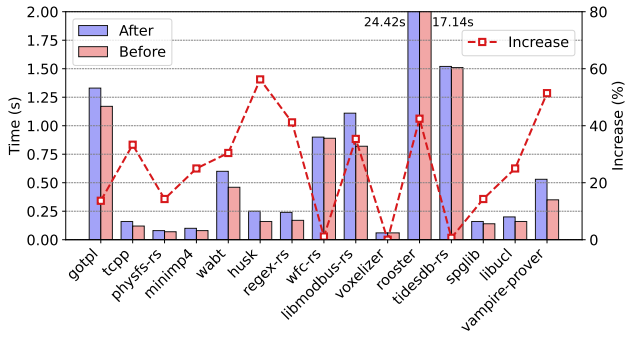
To answer RQ1, we first apply FFIPROBE to the dataset, which generates 44 bug warning reports. Second, we manually validate the authenticity of these reports and finally confirm 43 actual bugs across 29 packages, comprising 28 memory leaks, 14 instances of undefined behavior, and 1 out-of-bounds access. Among these confirmed bugs, the bugs in 11 packages are introduced for unit testing purposes. Additionally, three other packages explicitly acknowledged memory leaks in comments but indicated no plans for fixing them. The bugs in the remaining 15 packages are detailed in Table II, encompassing a total of 29 bugs. We have reported these issues to the respective package maintainers. As of the time of writing, two bugs have been confirmed and fixed.

Regarding the false positives, we conduct further analysis and reveal that they are primarily caused by static allocations for specific purposes. For example, functions such as `setlocale` in the C standard library allocate internal buffers that are not explicitly freed but are instead reclaimed by the operating system upon program termination, leading FFIPROBE to identify them as memory leaks.

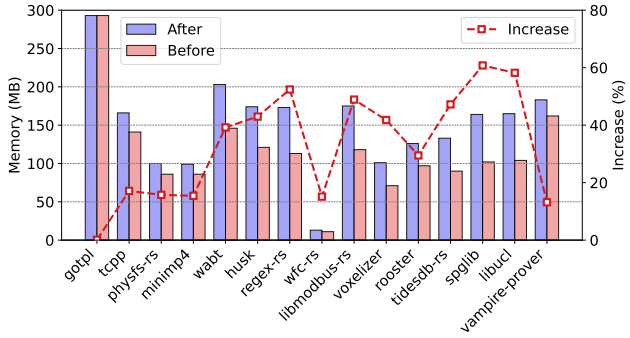
Moreover, the entire evaluation process is fully automated. Although the dataset includes various foreign languages and even binary modules, the testing process does not require any manual intervention. In summary, FFIPROBE can effectively detect Rust multilingual memory safety bugs without relying on foreign language source code.

C. RQ2: Performance

To answer RQ2, we measure the detection time and peak memory overhead for each package listed in Table II. To ensure result stability, we executed the detection five times for each package and calculated the averages. As shown in the sixth and eighth columns of the table, the average detection time is 2.11 seconds, and the average peak memory overhead is 152 MB. Furthermore, to assess the resource overhead introduced by instrumentation and the custom heap allocator, we compared the performance of the original programs against the instrumented versions, as illustrated in Fig. 4. The comparison reveals that instrumentation introduces an average runtime overhead of only 25.6% and a memory overhead of 33.2%. These results demonstrate that FFIPROBE



(a) Time cost



(b) Memory overhead

Fig. 4: Comparison of performance overhead before and after instrumentation

can effectively detect multilingual memory bugs in Rust with reasonable overhead.

IV. Related Work

Rust Dynamic Analysis. Prior studies have presented numerous dynamic analysis approaches and uncovered a wide variety of bugs. Jung et al. [14] proposed Miri, which implements a Rust abstract machine and utilizes this machine to interpret and execute MIR for detecting undefined behavior in Rust programs. Cho et al. [15] introduced RustSan, an address sanitizer tailored for Rust inspired by AddressSanitizer [20], which leverages cross-IR analysis to perform selective instrumentation on Rust MIR for detecting safety issues. Liu et al. [16] presented XRust, a tool centered around a novel heap allocator that isolates unsafe objects from safe ones in distinct memory regions, thereby preventing memory corruption across these regions.

Multilingual Bug Detection. There have been numerous works on multilingual bug detection in Rust. To the best of our knowledge, FFIChecker [13] is the pioneering work in this area and employs abstract interpretation and leverages taint analysis to analyze program variables. However, it solely focuses on Rust objects and only analyzes foreign C/C++ code when necessary. Hu et al. [12] present CRUSTIR and port several existing Rust analysis tools to its IR, enabling them to analyze multilingual programs. McCormack et al. [8] conducted an empirical study on undefined behavior in multilingual Rust programs, using Miri and an LLVM interpreter to jointly execute programs. Our work is closely related to theirs, but we adopt an instrumentation-based approach and are capable of detecting a broader range of bug types. Schuermann et al. [21] proposed Omniglot, which employs sandboxing techniques to invoke FFIs and introduces a specialized type to facilitate safe access to foreign memory within the sandbox. Schuermann et al. [17] proposed Encapsulated Functions, which leverage hardware-based memory protection mechanisms to restrict the access of unsafe code to Rust's safe memory.

V. Conclusion

In this work, we present an approach for detecting Rust multilingual memory bugs through dynamic analysis. Our approach leverages Rust MIR as the static instrumentation target, instrument probe functions at the multilingual interaction boundaries via the instrumentation algorithm we designed. We also design the probe functions according to the Rust multilingual memory bugs. To assist the probe functions in identifying safety bugs, we design a custom heap allocator to proxy memory allocation requests and maintain memory object metadata. We implement a prototype for FFIProbe and conduct experiments to evaluate its effectiveness and performance. The experimental results demonstrate that FFIProbe can effectively detect Rust memory safety bugs caused by multilingual interactions in real-world Rust programs, while maintaining a favorable resource overhead. 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] 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.

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

[4] 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*. Association for Computing Machinery, May 2016, pp. 81–89.

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

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

[7] Z. Li, V. Narayanan, X. Chen, J. Zhang, and A. Burtsev, "Rust for linux: Understanding the security impact of rust in the linux kernel," in *2024 Annual Computer Security Applications Conference (ACSAC)*. IEEE, Dec. 2024, pp. 548–562.

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

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

[10] 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*. Association for Computing Machinery, Oct. 2020, pp. 246–257.

[11] 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*. ACM, Apr. 2024, pp. 1–13.

[12] 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.

[13] 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*. Springer-Verlag, Sep. 2022, pp. 680–700.

[14] R. Jung, B. Kimock, C. Poveda, E. S. Muñoz, O. Scherer, and Q. Wang, "Miri: Practical undefined behavior detection for rust," *Artifact for Miri: Practical Undefined Behavior Detection for Rust*, vol. 10, no. POPL, pp. 48:1383–48:1411, Jan. 2026.

[15] K. Cho, J. Kim, K. D. Duy, H. Lim, and H. Lee, "Rustsan: Retrofitting addresssanitizer for efficient sanitization of rust," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 3729–3746.

[16] 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*. ACM, Jun. 2020, pp. 234–245.

[17] L. Schuermann, A. Thomas, and A. Levy, "Encapsulated functions: Fortifying rust's ffi in embedded systems," in *Proceedings of the 1st Workshop on Kernel Isolation, Safety and Verification*. Association for Computing Machinery, Oct. 2023, pp. 41–48.

[18] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang, "Understanding memory and thread safety practices and issues in real-world rust programs," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 2020, pp. 763–779.

[19] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "Rustbelt: Securing the foundations of the rust programming language," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 66:1–66:34, Dec. 2017.

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

[21] 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.