ACORN: Towards a Holistic Cross-Language Program Analysis for Rust

Lei Xia, Baojian Hua, and Yang Wang
School of Software Engineering, University of Science and Technology of China, China
Suzhou Institute for Advanced Research, University of Science and Technology of China, China
xialeics@mail.ustc.edu.cn bjhua@ustc.edu.cn angyan@ustc.edu.cn

Abstract—Despite the fact that Rust is designed to be the next generation of safe language for system programming, Rust programs are still vulnerable and exploitable due to its inclusion of an `unsafe` sub-language for programming flexibility and efficiency. Unfortunately, existing studies on Rust security mostly focus on pure Rust code but ignore multilingual Rust applications interacting with unsafe external code such as C. As a result, vulnerabilities *at* and *across* language boundaries are largely left out as blind spots.

In this paper, to fill the gap, we present ACORN, a holisticwobazh program analysis framework for multilingual Rust programs. Our key idea is to utilize a low-level, expressive, and language-independent intermediate representation as the unified specification language. Then by translating both Rust and C into this unified specification, we can eliminate language boundaries and perform holistic program analyses to detect cross-language vulnerabilities. Based on this key idea, we have leveraged WebAssembly, a novel low-level code format designed for *execution*, as the unified specification for *analysis*. We then designed and formalized conversions from both Rust and C to this unified specification, and implemented analysis algorithms to detect vulnerabilities. We have implemented a software prototype for ACORN and conducted extensive experiments to evaluate its effectiveness and performance. Experimental results demonstrated that ACORN can effectively detect vulnerabilities in multilingual Rust programs with negligible overhead (0.06 seconds for each test case on average), and it outperforms state-of-the-art peer tools such as FFIChecker and Rudra.

Keywords–Rust, Security, Multilingual Program Analysis

1. INTRODUCTION

Safe programming languages are essential in developing reliable and trustworthy software infrastructures. While C/C++ have traditionally been the dominant languages for building such infrastructures such as operating system kernels, network stacks, and DBMSs, they are prone to causing serious and subtle bugs due to their unsafe nature. Rust [1] is an emerging language specifically designed for secure system programming. Rust not only provides strong security guarantees through a unique set of language features focused on safety (*e.g.*, ownership [2], and lifetime [3]), but also maintains high efficiency comparable to that of C/C++, by embracing a zero-cost abstraction philosophy [4]. Due to its technical advantages, Rust is gaining rapid adoptions in many scenarios such as system kernels [5] [6] [7], Web browsers [8], and language runtime [9].

While Rust has made a significant step towards safe system programming, it is not a panacea. Specifically, for programming flexibility and efficiency, Rust has incorporated a sub-language dubbed as *unsafe Rust* [10], which might lead to serious vulnerabilities due to the lacking of both static and runtime checkings. On the one hand, *unsafe Rust* does not perform any static security checking, hence unsafe operations (*e.g.*, arbitrary pointer casting) are not detected. On the other hand, *unsafe Rust* does not provide dynamic checking, hence notorious bugs (*e.g.*, buffer overflow [11]) may still manifest. Consequently, *unsafe Rust* introduces a security loophole which has already led to serious bugs and vulnerabilities [12] [13].

Recognizing this problem, a considerable amount of security studies has been conducted on Rust (*e.g.*, vulnerability detection [14] [15] [16] [17], and vulnerability prevention [18] [19] [20]). Unfortunately, existing studies mostly focus on pure Rust code, but ignore the vulnerabilities which might arise in multilingual Rust applications interacting with external unsafe code such as C through Foreign Function Interface (FFI) [21]. The key difficulty in analyzing multilingual Rust application lies in the fact that vulnerabilities can exist both *at* and *across* language boundaries [22], rather than in a single language. Hence, a holistic program analysis framework is critical and essential for enhancing the security of multilingual Rust applications.

Yet despite this criticality and essentiality, developing a holistic analysis for multilingual Rust systems faces two challenges: **C1**) language disparity; and **C2**) cost-effectiveness. First, Rust and other languages have different language features (*e.g.*, semantics, and memory models), making a holistic analysis challenging. For example, CRust [23] utilized MIR [24], an intermediate representation in Rust compiler, to eliminate language disparities between Rust and C. However, translation from C to MIR is difficult as MIR does not model the semantics of C honestly. Consequently, this approach is still unsatisfactory as it is not language-agnostic. Second, to be practically useful, the holistic analysis should be cost-effective by scaling to real-world applications, hence, heavy-weight solutions are not ideal due to their lack of scalability. For example, FFIChecker [25] relied on additionally annotating FFIs on LLVM IR [26], which is not only complex but also error-prone.

To address these challenges, we argue that a comprehensive and cost-effective specification language is necessary for

analyzing multilingual Rust systems. Specifically, we argue that the specification language should satisfy the following four requirements: **R1**) *expressiveness*: it should be expressive enough to represent diverse programming languages features; **R2**) *neutrality*: to provide unbiased analysis, the specification language should be source-language independent without favoring any particular one; **R3**) *formality*: formally-defined and unambiguous semantics are essential to precisely capture the programs behaviors and ensure the reliability of analysis; and **R4**) *rich analysis support*: the specification language's rich analysis support will reduce complexity and efforts to develop holistic analysis.

In this paper, we propose ACORN, the first holistic cross-language program analysis for multilingual Rust. Our key idea is to leverage WebAssembly (Wasm) [27], a novel low-level code format designed for *execution*, as the specification for the holistic cross-language program *analysis*. With this key idea, we first formalized a core subset of Wasm as the unified specification language. Then, we designed and formalized conversion from multilingual Rust programs to Wasm. Finally, we implemented program analysis algorithms on Wasm to detect security vulnerabilities manifested in multilingual Rust programs.

We argue that our approach satisfies the aforementioned four requirements for the specification language. *First*, Wasm has expressive language features, making it a promising target for compilation of static (*e.g.*, Rust [28], and C/C++ [29]) or dynamic (*e.g.*, Python [30], and JavaScript [31]) languages. *Second*, Wasm serves as portable compilation target for high-level programming languages, making it inherently source-independent. *Third*, Wasm provides well-defined and formal specification precisely describing the behavior of the binary instructions. *Finally*, Wasm is designed to have intrinsic support (*e.g.*, structured control flows, and a strong type system) for program reasoning and analysis, making program analysis feasible [32] [33] [34] [35].

To validate our design, we have implemented a prototype for ACORN, and have conducted extensive experiments to evaluate it in terms of the effectiveness and performance. First, we evaluate the effectiveness of ACORN by applying it on two benchmarks: 1) a micro-benchmark containing common kinds of security vulnerabilities, and 2) a real-world benchmark collected from CWE [36]. Experimental results demonstrated that ACORN can effectively detect vulnerabilities in multilingual Rust programs with an negligible overhead. Second, comparison with state-of-the-art shows that ACORN is superior to peer tools such as Rudra [15] and FFIChecker [25], with respect to capabilities of vulnerabilities detection.

Contributions. To summarize, this work represents a first step towards designing a holistic program analysis for multilingual Rust applications, and thus makes the following contributions:

- **A holistic program analysis ACORN.** We present ACORN, the first holistic program analysis for multilingual Rust programs.
- **A prototype implementation of ACORN.** We implemented a prototype of ACORN for multilingual Rust programs.
- **Extensive evaluation.** We conducted extensive experiments to evaluate the effectiveness and performance of ACORN.

Outline. The rest of this paper is organized as follows. We first introduce the background (§ 2), and our motivation (§ 3). Next, we present the core syntax of the specification language (§ 4), and formally defined translation rules (§ 5). We then introduce the implementation (§ 6) and evaluations (§ 7). Finally, we discuss limitations (§ 8), related work (§ 9), and conclude (§ 10).

2. Background

To be self-contained, this section presents the necessary background knowledge on Rust (§ 2-A), Wasm (§ 2-B), and multilingual program analysis (§ 2-C).

2.1. Rust

Capsule history. Rust is an emerging programming language designed for building reliable and efficient system software. It originated as a personal project by Graydon Hoare in 2006 and was later officially sponsored by Mozilla in 2009 [37]. Rust 1.0 was released in 2015, marking a stable and production ready version of the language, and its latest stable version is 1.68.2 (as of this study). With over 15 years of active development, Rust is becoming more mature and productive.

Advantages. Rust emphasizes security and performance. First, Rust provides safety guarantees via a unique ownership and borrowing system [2], alongside a sound type system [38] based on linear logic [39] and alias types [40]. These advanced language features not only rule out memory vulnerabilities such as dangling pointers, memory leaking, and double frees, but also enforce thread safety by preventing data races and deadlocks [4]. Second, Rust achieves high efficiency through an ownership-based explicit memory management and a lifetime model, without any garbage collectors [41]. Both the ownership and lifetime are checked and enforced at compile-time, thus incurring zero runtime overhead.

Wide adoptions. Rust has been widely adopted across diverse domains in recent years. For example, Rust has been used successfully to build software infrastructures, such as operating system kernels [5] [6] [7], Web browsers [8], network protocol stacks [42], language runtime [9], databases [43], and blockchains [44]. Moreover, Rust is gaining more adoptions in the industry (*e.g.*, Microsoft [45], Google [46], and even Linux [47]).

2.2. Wasm

Brief history. Wasm was introduced by Google and Mozilla in 2015 [48] and quickly gained popularity, becoming a de facto standard language in browsers by 2017 [49]. In 2018, the first complete formal definition of Wasm was released [50], solidifying its specifications. The W3C officially recognized Wasm as the fourth Web standard in 2019 [51]. Over time, Wasm has evolved and matured, with the development of the WebAssembly System Interface (WASI) [52] and the ongoing work on the standard version 2.0 draft [50]. Today, Wasm is a

stable and production-quality language that finds applications in both Web and standalone environments.

Advanced features. Wasm is designed with a focus on safety, efficiency, and portability [53]. First, to ensure program safety, Wasm incorporates secure features such as strong typing, sandboxing isolation and control flow integrity [32] [54]. Second, Wasm's virtual machine (VM) is optimized for space usage and execution performance, allowing it to leverage hardware capabilities effectively across different platforms. Third, WASI provides a standardized and safe system interaction interfaces, enabling Wasm programs to be deployed outside of Web browsers.

Applications. Wasm's advanced features have contributed to its widespread adoption in both Web and non-Web domains. In the Web domain, Wasm became the fourth official language (after HTML, CSS, and JavaScript) fully supported by major browsers. In non-Web domains, Wasm has been adopted in a wide range of computing scenarios, such as cloud computing [55] [56], IoT [57], blockchain [58] [59] [60] [61], edge computing [62], video transcoder [63], and game engines [64]. In the future, the growing need to secure cloud and edge computing infrastructures while maintaining high efficiency will drive Wasm to become an even more promising language.

2.3. Multilingual Programming

A multilingual program refers to the practice of using multiple programming languages within a single software system. Multilingual programming enables developers to take advantage of the strengths of different languages, or to reuse lagacy libraries or code developed in different languages, and thus is widely used in many software systems such as Mozzila [65], PyTorch [66] and NumPy [67]. To support seamlessly interoperation between different languages, FFIs are common mechanisms to connect different languages (*e.g.*, Java's Native Interface (JNI) [68], and Rust and Python's C FFI [21] [69]).

3. Motivation

In this section, we present the motivation for this work by first discussing challenges in analyzing multilingual Rust programs, then introduce our key idea to address these challenges.

3.1. Challenges in Analyzing Multilingual Rust

Analyzing multilingual Rust programs is challenging because vulnerabilities may arise *at* or *across* language boundaries, instead of in a single language.

To better illustrate the challenges, we present, in Figure 1, sample multilingual Rust programs consisting of serious memory vulnerabilities: double-free, use-after-free, and buffer overflows.

Double-free. A double-free (DF) is a serious memory bug which might lead to unpredictable behaviors or even crashes. Figure 1(a) presents a DF bug: first, the Rust variable `n` is passed to the C function (line r5), which is then reclaimed by C (line c4); then the variable `n` is automatically deallocated for a second time when it goes out of its lexical scope (line r7), triggerring a DF vulnerability.

Figure 1: Sample multilingual Rust programs illustrating three kinds of memory vulnerabilities across Rust and C: a double-free in (a), a use-after-free in (b), and a buffer overflow in (c).

Use-after-free. A use-after-free (UaF) occurs when a program uses a memory cell after it has been deallocated. Figure 1(b) depicts a UaF bug in a multilingual Rust program: the Rust vector `heap_obj` is passed to a C function `c_fun` (line r5), which is then reallocated (line c6); then subsequent access to the variable in Rust (line r8) will triggers a UaF bug.

Buffer overflow. A buffer overflow (BO) occurs when a program writes more data exceeding a buffer's capacity. Figure 1(c) presents a BO bug in a multilingual Rust program: the Rust variable `buffer` is passed as an argument to the C function (line r5), which is written beyond the buffer length (line c6), triggering a BO bug (*i.e.*, a notorious off-by-one bug).

These sample programs not only illustrate how bugs may manifest in multilingual applications, but also justify holistic program analysis, as any single language analysis will miss these bugs due to their lack of cross language information. In the meanwhile, we cannot just ignore these multilingual programs as FFI invocations are the most pervasive `unsafe` operations in Rust, accounting for 22.5% of all `unsafe` function calls [13] and more than 72% of packages on the official Rust package registry (crates.io [70]) depending on at least one unsafe FFI-bindings package [25]. Recent empirical studies [12] [13] have shown that the incorrect use of FFI is one of the primary causes of real world memory safety bugs. Furthermore, while we have concentrated on memory vulnerabilities here, the discussion and conclusion also applied to other vulnerabilities such as concurrency bugs (see § 8).

3.2. Our Key Idea

To address these challenges, our key idea is to leverage a unified specification language to enable a holistic program analysis for multilingual Rust programs. With this unified specification language, we can translate both Rust and C

in multilingual Rust to it, hence eliminating the language boundaries. As as result, we can implement program analysis algorithms on this unified specification language, just as for single language. Hence, the key problem to address is to design a unified specification language, which we will discussed in the next section.

4. A Unified Specification Language

Given our key idea for the specification language, two key questions remain: (1) "Which specification language is most suited as a unified analysis platform for multilingual Rust programs?" and (2) "How to convert multilingual Rust programs to such a specification language?"

In this section, we answer these questions by first presenting detailed requirements for specification language (§ 4-A), then introduce the syntax (§ 4-B). We illustrate how to formalize conversions from both Rust and C languages to this specification in the next section (§ 5).

4.1. Requirements for Specification Language

We argue a unified specification language for multilingual program analysis should satisfy the following four requirements:

Expressiveness. It should be expressive enough to capture the diverse semantics and behaviors of different programming languages, such as control flow patterns, data structures, and type systems. Thus, the specification can precisely model different programming languages.

Neutrality. Impartiality towards source code is another crucial aspect of the specification language. It should provide a neutral ground for faithful representation and analysis of multilingual Rust programs, which will enhance the objectivity of the analysis.

Formality The specification must possess well-defined and unambiguous semantics to provide clear and unequivocal interpretations of program behaviors. This precision in semantics ensures accurate analysis results, ensuring the reliability and reproducibility of the analysis.

Analysis Support. The intrinsic supports provided by the specification, such as strong typing, structured control flow, and explicit memory management, possessed by the specification for analysis will making program analysis much more practical.

To this end, we propose that Wasm, a novel low-level code format designed for *execution*, is the ideal unified specification for *analysis*, which satisfies the aforementioned four requirements: 1) Wasm has a rich language support with diverse programming languages actively adopting or exploring their compilation to it. Rust [28], C/C++ [29] have full support for compiling to Wasm. Python [30], JavaScript [31] and Go [71] provide partial support, enabling compilation of specific features to Wasm; 2) Wasm is designed to be a portable compilation target for programming languages, which makes Wasm inherently source-independent; 3) Wasm aims to establish a secure and reliable execution environment, thus, it provides well-defined and formal specification that precisely describes the behavior of the binary instructions; and

Val. Type	ρ	$::=$	`i32` | `i64` | `f32` | `f64`
Func. Type	σ	$::=$	$\rho^* \to \rho^*$
Type	τ	$::=$	$\rho \mid \sigma$
Binary Op.	b	$::=$	ρ.`add` | ρ.`mul` | ρ.`shl` | ...
Unary Op.	u	$::=$	ρ.`abs` | ρ.`eqz` | ...
Load/Store	l	$::=$	ρ.`load` | ρ.`store`
Local Op.	c	$::=$	`local`.(`set` | `get`) x
Global Op.	g	$::=$	`global`.(`set` | `get`) x
Call	t	$::=$	`call` f | `call_indirect` σ
Instr.	i	$::=$	$b \mid u \mid l \mid c \mid g \mid t$
		|	`drop` | `nop` | `if` | `else` | `block`
		|	`loop` | `end` | `br` a | `br_if` a
		|	`select` | ρ.`const` c | ...
Function	f	$:=$	$\sigma\ x\{i^*\}$
Module	m	$:=$	f^*

Figure 2: Core syntax of Wasm language.

4) inherent supports provided by Wasm, including features like structured control flow, and the robust type system, facilitate program reasoning and analysis, rendering program analysis practicable [32].

Furthermore, extensive researches have been conducted in the field of program analysis on Wasm, with several noteworthy studies making substantial contributions in this domain [33] [34] [35] [72] [73]. These studies offer valuable algorithms and techniques that can be harnessed for cross-language analysis grounded in Wasm, thereby streamlining the implementation of the framework we proposed.

It should be noted that, unlike LLVM [26], Wasm is *not* designed to be a program analysis and optimization framework, rather, it serves as an *execution* format by Wasm virtual machines. Hence, our work stands as a pioneering effort in exploring its potential for program analysis. This endeavor holds the promise of benefiting the broader community engaged in multilingual program analysis.

4.2. Syntax

To formalize the unified specification language, we present, in Figure 2, the core syntax of Wasm via a context-free grammar. Each Wasm module m consists of a list of functions f, whose body contains a sequence of instructions i. A function f may have multiple arguments and return results, indicated by its type $\rho^* \to \rho^*$ (the notation * stands for a Kleene closure).

An instruction i consists of binary/unary operations b or u, memory load/stores l, structured control flows `if` or `loop`, and function invocation/return t (some irrelevant instructions are omitted for brevity). Wasm instructions demonstrated three distinct properties: first, Wasm is a stack-based VM in that operands and the result of an operation are always on top of the operand stack. For example, the addition operation `i32.add` pops two operands from the operand stack and pushes the result. Second, Wasm instructions are strongly typed in specifying the expected type in the opcode (*e.g.*, the

Bid	b	$\in$	$\mathbb{Z}$
Type	τ	$::=$	$\texttt{bool} \mid \texttt{i32} \mid \texttt{unit}$
Operand	o	$::=$	$\texttt{const}\ c \mid \texttt{move}\ x \mid \texttt{copy}\ x$
Rvalue	r	$::=$	$o \mid o_1 + o_2$
Statement	s	$::=$	$x = r \mid s_1; s_2$
Terminator	t	$::=$	$f(x) \mid \texttt{return} \mid \texttt{goto}(b) \mid \texttt{drop}(x)$
Function	f	$::=$	$\texttt{fn}\ z(\overrightarrow{x : \tau}) \rightarrow \tau\ \{\overrightarrow{y : \tau}; s; t\}$

Figure 3: Core syntax of the Rust language.

Type	τ	$::=$	$\texttt{bool} \mid \texttt{int} \mid \texttt{void}$
Value	v	$::=$	$\texttt{true} \mid \texttt{false} \mid \texttt{n}$
Expression	e	$::=$	$v \mid x \mid e_1 + e_2$
Statement	s	$::=$	$x = e \mid s_1; s_2 \mid \texttt{if}(e)\ s_1\ s_2$
		$\mid$	$\texttt{while}(e)\ s \mid \texttt{return}(e)$
Function	f	$::=$	$\tau\ z(\overrightarrow{\tau\ x})\ \{\overrightarrow{\tau\ y}; s\}$

Figure 4: Core syntax of the C language.

i32 prefix in the i32.abs instruction), facilitating binary-level type checking and analysis. Third, Wasm supports *structured* control flows (*e.g.*, if or loop), making compilation to Wasm easier.

5. Translation Models

In this section, we present translation rules from both Rust and C to this specification language.

5.1. Formalizing Rust and C

We first formalizing both Rust and C, for subsequent translations.

Formalizing Rust. We introduce a simplified language that captures a subset of Rust MIR syntax in Figure 3, to present the translation rules intuitively and rigorously. Given the complex nature of Rust, attempting to model all language features would be impractical, thus we only extract necessary components useful for our purposes.

Specifically, A Rust program consists of several functions, each function f consists a list of parameters $\overrightarrow{x : \tau}$, a return type τ, a list of local variable declarations $\overrightarrow{y : \tau}$, followed by statements s and a terminator t. A statement s may be an assignment $x = r$, or recursively generate two statements $s_1; s_2$. Like the control-flow graph in compilers, controls in function f can only exit from the terminator t, which has several distinct syntactic forms: 1) function calls $f(x)$ and returns `return`; 2) an unconditional jump `goto` and 3) a deletion `drop` which explicitly deallocates the memory of x. A right value r including operands o and the addition of operand o_1 and o_2. Operands o have Rust specific syntactic features: `move` and `copy`, which represents the move [74] or copy [75] semantics in Rust, respectively. A type τ consists of representative Rust types, including three primitive types: `bool`, `i32` and `unit`, where `unit` is a Rust-specific concept used to represent the absence of a meaningful value.

Formalizing C. Figure 4 outlines the syntax for a subset of C including fundamental features of C. This subset encompasses basic types τ and values v, expressions e for calculations, statements s for assignment (e.g., $x = e$) and program flow control (such as `if` and `while`), as well as functions f for code organization.

While some other features such as union in C and array in Rust have been omitted, they can be integrated without encountering technical complexities.

5.2. Compiling Rust and C to Wasm

The rules for compiling Rust and C to Wasm are formally defined through a set of judgments. We use the notation $\overrightarrow{I}$ for a sequence of Wasm instructions, and Φ to represent a compilation environment constructed by the compiler. We use $[\,]$ to denote an empty sequence, and @ for concatenation of instructions. For example, $\overrightarrow{I_1} @ \overrightarrow{I_2}$ denotes $\overrightarrow{I_1}$ and $\overrightarrow{I_2}$ will be executed in sequence.

Compiling Rust. Figure 5 presents the rules for compiling key features of Rust into a sequence of Wasm instructions $\overrightarrow{I}$. First, before compiling Rust functions, the compiler will construct a compilation environment Φ, which is formalized with three auxiliary functions mapTy(-), mapParm(-), and mapVar([-]). The function mapTy(τ) maps a Rust type τ to a Wasm type ρ. Due to the streamlining of Wasm's type system, All three Rust types will be mapped to i32 in Wasm. For example, we use the i32 type to represent bool where 0 stands for false and 1 stands for true. Likewise, 0 of i32 type is utilized to represent the Rust unit type which signifies the absence of a meaningful value.

$$\begin{aligned} \text{mapTy}(\texttt{bool}) &= \texttt{i32} \\ \text{mapTy}(\texttt{i32}) &= \texttt{i32} \\ \text{mapTy}(\texttt{unit}) &= \texttt{i32} \end{aligned}$$

The function $\text{mapParm}([x_1 : \tau_1; \ldots])$ maps a Rust function parameter to a Wasm type τ to represent this parameter:

$$\text{mapParm}([x_1 : \tau_1; \ldots]) = ([\text{mapTy}(\tau_1); \ldots])$$

The function $\text{mapVar}([y_1 : \tau_1; \ldots; y_n : \tau_n], d)$ maps Rust variable declarations in Rust function body to Wasm local variables starting from the index d_1 as follows:

$$\text{mapVar}([y_1 : \tau_1; \ldots], d_1) = [y_1 : (d_1, \text{mapTy}(\tau_1); \ldots]$$

The Rust function body s and t are then compiled to $\overrightarrow{I_s}$ and $\overrightarrow{I_t}$ under the environment Φ, respectively; followed by return type ρ_{ret}, which is mapped from the return type τ of the Rust function.

Next, we use the following two judgments to compile statements s and terminators t, respectively:

$$\Phi \vdash s \rightsquigarrow \overrightarrow{I} \qquad \Phi \vdash t \rightsquigarrow \overrightarrow{I}$$

A statement s is compiled into a list of Wasm instructions $\overrightarrow{I}$, as does for an expression e. For example, to compile "$x = r$", we first compile the rvalue r, then the result of rvalue r which

$$\boxed{\Phi \vdash f \rightsquigarrow \vec{I}}$$

$$\frac{\Phi = \{\text{mapParm}([\overrightarrow{x : \tau}]) \quad \text{mapVar}([\overrightarrow{y : \tau}], 0)\} \qquad \rho_{\text{ret}} = \text{mapTy}(\tau) \quad \Phi, \rho_{\text{ret}} \vdash s \rightsquigarrow \vec{I_s} \quad \Phi, \rho_{\text{ret}} \vdash t \rightsquigarrow \vec{I_t}}{\vdash \left(\begin{array}{c} \texttt{fn}\ z(\overrightarrow{x : \tau}) \to \tau \\ \{\overrightarrow{y : \tau}; s; t\} \end{array} \right) \rightsquigarrow \rho_x^* \to \rho_{ret}\ z(\vec{I_s}; \vec{I_t})}$$

$$\boxed{\Phi \vdash s \rightsquigarrow \vec{I}}$$

$$\frac{\Phi(x) = (d, \rho) \qquad \Phi \vdash r \rightsquigarrow (\vec{I}, \rho)}{\Phi \vdash x = r \rightsquigarrow \vec{I} @ [\rho.\texttt{store}\ d]}$$

$$\frac{\Phi \vdash s_1 \rightsquigarrow \vec{I_1} \qquad \Phi \vdash s_2 \rightsquigarrow \vec{I_2}}{\Phi \vdash s_1; s_2 \rightsquigarrow \vec{I_1} @ \vec{I_2}}$$

$$\boxed{\Phi \vdash t \rightsquigarrow \vec{I}}$$

$$\frac{}{\Phi \vdash \texttt{return} \rightsquigarrow [\texttt{return}]}$$

$$\frac{}{\Phi \vdash \texttt{goto}(b) \rightsquigarrow [\texttt{br}\ \mathit{label}\ b]}$$

$$\frac{\Phi(x) = (d, \rho)}{\Phi \vdash \texttt{drop}(x) \rightsquigarrow ([\rho.\texttt{load}\ d], \rho) @ [\texttt{call}\ \mathit{free}]}$$

$$\frac{\Phi \vdash f \rightsquigarrow \vec{I} \qquad \Phi(x) = (d, \rho)}{\Phi \vdash f(x) \rightsquigarrow ([\rho.\texttt{load}\ d], \rho) @ [\texttt{call}\ f] @ \vec{I}}$$

$$\boxed{\Phi \vdash r \rightsquigarrow (\vec{I}, \rho)}$$

$$\frac{\Phi \vdash o_1 \rightsquigarrow (\vec{I_1}, \rho) \qquad \Phi \vdash o_2 \rightsquigarrow (\vec{I_2}, \rho)}{\Phi \vdash o_1 + o_2 \rightsquigarrow (\vec{I_1} @ \vec{I_2} @ ([\rho.\texttt{add}], \rho))}$$

$$\boxed{\Phi \vdash o \rightsquigarrow (\vec{I}, \rho)}$$

$$\frac{}{\Phi \vdash \texttt{const}\ c \rightsquigarrow ([\texttt{i32.const}\ c], \texttt{i32})}$$

$$\frac{\Phi(x) = (d, \rho)}{\Phi \vdash \texttt{move}\ x \rightsquigarrow ([\rho.\texttt{load}\ d], \rho)}$$

$$\frac{\Phi(x) = (d, \rho)}{\Phi \vdash \texttt{copy}\ x \rightsquigarrow ([\rho.\texttt{load}\ d], \rho)}$$

Figure 5: Rules for Compiling Rust to Wasm.

has been on the top of Wasm stack, will be stored to the variable x of type ρ at memory offset d. Specially, `move` x will transfer ownership whereas `copy` x will not, however, they both evaluate the value of x from the operational semantics point of view.

Finally, we show rules for compiling operands and rvalues, which are formalized by the following judgments, respectively:

$$\Phi \vdash o \rightsquigarrow (\vec{I}, \rho) \qquad \Phi \vdash r \rightsquigarrow (\vec{I}, \rho)$$

An operand o is compiled into a list of Wasm instructions $\vec{I}$ that put the value of the operand on the top of the Wasm stack, and also returns the Wasm type ρ of the value. An example is the rule for a constant c, in which it just pushes the constant onto the stack. Rules for compiling rvalues are similar.

$$\boxed{\Phi \vdash f \rightsquigarrow \vec{I}}$$

$$\frac{\Phi = \{\text{mapParm}([\overrightarrow{\tau\ x}]) \quad \text{mapVar}([\overrightarrow{\tau\ y}], 0)\} \qquad \rho_{\text{ret}} = \text{mapTy}(\tau) \qquad \Phi, \rho_{\text{ret}} \vdash s \rightsquigarrow \vec{I_s}}{\vdash \left(\begin{array}{c} \tau\ z(\overrightarrow{\tau\ x}) \\ \{\overrightarrow{\tau\ y}; s; \} \end{array} \right) \rightsquigarrow \rho_x^* \to \rho_{ret}\ z(\vec{I_s})}$$

$$\boxed{\Phi \vdash s \rightsquigarrow \vec{I}}$$

$$\frac{\Phi(x) = (d, \rho) \qquad \Phi \vdash e \rightsquigarrow (\vec{I}, \rho)}{\Phi \vdash x = e \rightsquigarrow \vec{I} @ [\rho.\texttt{store}\ d]}$$

$$\frac{\Phi \vdash s_1 \rightsquigarrow \vec{I_1} \qquad \Phi \vdash s_2 \rightsquigarrow \vec{I_2}}{\Phi \vdash s_1; s_2 \rightsquigarrow \vec{I_1} @ \vec{I_2}}$$

$$\frac{\Phi \vdash e \rightsquigarrow (\vec{I_e}, \rho) \quad \Phi \vdash s_1 \rightsquigarrow \vec{I_1} \quad \Phi \vdash s_2 \rightsquigarrow \vec{I_2}}{\Phi \vdash \texttt{if}(e)\ s_1\ s_2 \rightsquigarrow \vec{I_e} @ [\texttt{if}] @ \vec{I_1} @ [\texttt{else}] @ \vec{I_2} @ [\texttt{end}]}$$

$$\frac{\Phi \vdash e \rightsquigarrow (\vec{I_e}, \rho) \quad \Phi \vdash s \rightsquigarrow \vec{I_s}}{\Phi \vdash \texttt{while}(e)\ s \rightsquigarrow \vec{I_e} @ [\texttt{loop}] @ \vec{I_s} @ [\texttt{end}]}$$

$$\frac{\Phi \vdash e \rightsquigarrow (\vec{I}, \rho)}{\Phi \vdash \texttt{return}(e) \rightsquigarrow \vec{I_e} @ [\texttt{return}]}$$

$$\boxed{\Phi \vdash e \rightsquigarrow (\vec{I}, \rho)}$$

$$\frac{}{\Phi \vdash n \rightsquigarrow ([\texttt{i32.const}\ n], \texttt{i32})}$$

$$\frac{\Phi(x) = (d, \rho)}{\Phi \vdash x \rightsquigarrow ([\rho.\texttt{load}\ d], \rho)}$$

$$\frac{\Phi \vdash e_1 \rightsquigarrow (\vec{I_1}, \rho) \qquad \Phi \vdash e_2 \rightsquigarrow (\vec{I_2}, \rho)}{\Phi \vdash e_1 + e_2 \rightsquigarrow (\vec{I_1} @ \vec{I_2} @ ([\rho.\texttt{add}], \rho))}$$

Figure 6: Rules for Compiling C to Wasm.

Compiling C. Figure 6 depicts rules for compiling C to Wasm, which are similar to that for Rust. First, we still use mapTy(-), mapParm(-) and mapVar([-]) to formalize the compilation environment Φ, respectively, which is constructed

by the compiler before compiling C functions, as follows:

$$\begin{aligned} \text{mapTy}(\texttt{bool}) &= \texttt{i32} \\ \text{mapTy}(\texttt{int}) &= \texttt{i32} \\ \text{mapTy}(\texttt{void}) &= \texttt{i32} \\ \text{mapParm}([\tau_1\ x_1; \ldots]) &= ([\text{mapTy}(\tau_1); \ldots]) \\ \text{mapVar}([\tau_1\ x_1; \ldots]) &= [\text{mapTy}(\tau_1)\ x_1; \ldots] \end{aligned}$$

Then, statements s will be compiled to sequences of Wasm instructions $\vec{I}$ intuitively through rules presented in Figure 6. As an example, to compile "$\texttt{if}(e)\ s_1 s_2$", we first compile the conditional expression e, statement s_1 and statement s_2 in sequence, and use the structured control instruction `if` supported by Wasm to synthesize the final result. Similarly, to compile "$\texttt{while}(e)\ s$", after compiling the expression e and statement s, we use the structured control instruction `loop` to finish the compilation.

Finally, we use the following judgment to compile an expression e:

$$\Phi \vdash e \rightsquigarrow (\vec{I}, \rho)$$

the expression e is compiled to a sequence of Wasm instructions $\vec{I}$ and return the type ρ of the value evaluated by the $\vec{I}$. Benefiting from the simplicity and compactness of the Wasm's syntax, most rules are straightforward.

6. Prototype Implementation

To valid our design and to conduct the evaluation, we have implemented a prototype for ACORN, consisting of two main components: (1) a converter for translating both Rust and C code to Wasm; and (2) a portion of existing program analysis algorithms and tools on Wasm. To implement the converter, we leveraged the compilation to Wasm officially supported by Rust and C. For the second component, we ported existing algorithms and tools so that they can process Wasm for cross-language program analysis. To be specific, we ported two state-of-the-art analysis algorithms and tools: 1) Wasmati [76], an efficient and versatile code property graph infrastructure; and 2) Wasabi [77], a framework devised for the dynamic analysis of Wasm programs, we designed and implemented customized security plugins for this dynamic analysis for vulnerability detections.

7. Evaluation

In this section, we conduct experiments to evaluate ACORN. We first present the research questions guiding the evaluation (§ 7-A), and the benchmarks (§ 7-C). Then, we present the experimental results (§ 7-D, § 7-E, and § 7-F). Finally, we present a case study demonstrating a real-world vulnerabilities detected by ACORN (§ 7-G).

7.1. Research Questions

To evaluate ACORN and present the experimental results, we mainly investigate the following research questions:

RQ1: Effectiveness. Is ACORN effective in detecting security vulnerabilities in multilingual Rust programs?

RQ2: Performance. What is the performance of ACORN?

RQ3: Comparison to peer tools. How does ACORN compare to state-of-the-art tools?

7.2. Experimental Setup

All experiments and measurements are performed on a server with one 4 physical Intel i5 core CPU and 4 GB of RAM running Ubuntu 20.04.

7.3. Datasets

We created two datasets to conduct the evaluation: 1) a micro-benchmarks and 2) a real-world benchmark with 73 test cases collected from CWE.

Micro-benchmark. Assessing the effectiveness of ACORN requires a multilingual benchmark suite equipped with ground truth for comprehensive analysis. However, at present, such a benchmark suite which can adequately test ACORN remains unavailable, and the task of curating ground truth for large or complex real-world programs may pose considerable challenges and may not be feasible. Thus, as shown in TABLE I, we manually create a micro-benchmark consisting of 20 test cases with 9 different types of vulnerabilities (as presented by the second column), including Buffer Overflow (BO), Use-after-Free (UaF), Double Free (DF), and so on.

Our focus for the micro-benchmark is on the coverage of diverse vulnerability types, and we will maintain and augment it by including more benchmarks and test cases covering more vulnerability types.

Real-world benchmark. We leveraged CWE [36] as our real-world benchmark, which is a set of 82 common "Weaknesses in Software Written in C", with a total of 117 example programs. To use CWE for the evaluation of ACORN, we first removed 48 example programs lacking necessary information such as incomplete data structures and undefined functions, then we added a Rust wrapper to each of the remaining 69 example programs in this CWE, turning them into Rust-C programs. Evaluating ACORN on well-established vulnerability sets like CWE demonstrates the effectiveness of ACORN on real-world multilingual Rust applications.

7.4. RQ1: Effectiveness of ACORN

To answer **RQ1** by demonstrating the effectiveness of ACORN, we successively applied ACORN to the micro-benchmark and the real-world CWE conduct experiments.

Results on Micro-Benchmark. As shown in the last column in TABLE I, ACORN successfully detected all vulnerabilities in this benchmark which can both be compiled by `rustc` and pass the detection of other three state-of-the-art Rust program analysis tools. Although the micro-benchmarks can not represent all real-world application scenarios, the experimental results on them still demonstrate ACORN's capacity to eliminate language disparities and effectiveness in detecting vulnerabilities in multilingual Rust applications.

Results on real-world CWE. TABLE II presents the number of *present* vulnerabilities (P) in this dataset, the absolute number of *true positive* (TP) and *false negative* (FN), as well as the *precision* (PI) and *recall* (RC).

TABLE I: Experimental results on the micro-benchmark containing the following vulnerability types: integer overflow (IO), use after free (UaF), double free (DF), tainted variable (TV), out of bounds (OOB), dangerous function (DFunc), buffer overflow (BOF), stack overflow (SOF) and format strings (FS).

Test Case	Vulnerability Type	WASM LOC	CT[1](s) / per line (ms)	AT[2](s) / per line (ms)	rustc	Miri	MirCHK[3]	Rudra	FFICHK[4]	ACORN (This work)
1	IO_1	25,433	0.061 / 0.002	3.18 / 0.125	✘	✘	✘	✘	✘	✔
2	IO_2	25,436	0.061 / 0.002	3.46 / 0.136	✘	✘	✘	✘	✘	✔
3	UaF_1	26,691	0.062 / 0.002	0.91 / 0.034	✘	✘	✘	✘	✘	✔
4	UaF_2	27,859	0.061 / 0.003	0.92 / 0.033	✘	✘	✘	✘	✔	✔
5	UaF_3	22,601	0.061 / 0.003	0.79 / 0.035	✘	✘	✘	✘	✔	✔
6	DF_1	22,488	0.064 / 0.002	0.74 / 0.033	✘	✘	✘	✘	✔	✔
7	DF_2	22,971	0.063 / 0.003	0.71 / 0.031	✘	✘	✘	✘	✔	✔
8	TV_1	27,768	0.060 / 0.003	0.39 / 0.014	✘	✘	✘	✘	✘	✔
9	TV_2	25,445	0.063 / 0.002	0.28 / 0.011	✘	✘	✘	✘	✘	✔
10	OOB_1	35,018	0.062 / 0.002	1.61 / 0.046	✘	✘	✘	✘	✘	✔
11	OOB_2	32,721	0.061 / 0.002	2.91 / 0.089	✘	✘	✘	✘	✘	✔
12	$DFunc_1$	35,019	0.060 / 0.002	0.42 / 0.012	✘	✘	✘	✘	✘	✔
13	$DFunc_2$	29,931	0.069 / 0.002	0.48 / 0.016	✘	✘	✘	✘	✘	✔
14	BOF_1	35,134	0.062 / 0.002	0.88 / 0.025	✘	✘	✘	✘	✔	✔
15	BOF_2	40,413	0.065 / 0.002	0.93 / 0.023	✘	✘	✘	✘	✔	✔
16	BOF_3	34,999	0.061 / 0.002	1.05 / 0.030	✘	✘	✘	✘	✔	✔
17	SOF_1	33,467	0.063 / 0.002	1.97 / 0.059	✘	✘	✘	✘	✘	✔
18	SOF_2	27,089	0.061 / 0.002	1.49 / 0.055	✘	✘	✘	✘	✘	✔
19	FS_1	29,931	0.067 / 0.002	1.23 / 0.041	✘	✘	✘	✘	✘	✔
20	FS_2	34,411	0.062 / 0.002	1.58 / 0.046	✘	✘	✘	✘	✘	✔

[1] "CT" means time for converting the source code to Wasm (Conversion Time).
[2] "AT" means time for program analysis on Wasm (Analysis Time).
[3] "MirCHK" means MirChecker: a static analysis tool for Rust.
[4] "FFICHK" means FFIChecker: a static analysis tool For detecting memory management bugs between Rust and C/C++.

TABLE II: Effectiveness results on real-world CWE.

Benchmark	P	TP	FN	PI	RC
CWE	73	65	8	100%	89.04%

For the total of 69 test cases, we first manually identified 73 vulnerabilities as ground truth. Among them, 65 were correctly reported (TP) and 8 were missed (FN) by ACORN, leading to a precision of 100% and a recall of 89.04%. The 8 false negatives divided into two categories: 1) 3 failed cases as Wasm lacking signal support, which result in them not being translated to Wasm; 2) the other 5 were unreported because the absence of corresponding detection algorithms including 4 logical errors and 1 numerical error; and the limitations of existing program analysis on Wasm including 1 stack buffer overflow.

Summary: ACORN achieved a 100% precision and a 89.04% recall on the real-world benchmark, respectively, demonstrating its effectiveness.

7.5. RQ2: Performance of ACORN

To answer **RQ2** by investigating the performance ACORN, we applied ACORN to the micro-benchmark and each of Rust-C program ran 10 rounds to calculate the average time.

TABLE I (the 4th and 5th columns) presents the performance of ACORN, including: 1) time for converting the source code to Wasm (**Conversion Time**); and 2) time for program analysis on Wasm (**Analysis Time**). Experimental results demonstrated that ACORN is efficient in conducting holistic program analysis in multilingual Rust applications: the time spent on code conversion into ACORN is about 0.06 seconds for each case (or

approximately 0.002 milliseconds per line), whereas the analysis time is around 1.26 seconds for each program (or 0.011 to 0.136 milliseconds per line). Furthermore, the conversion time is significantly less than the analysis time, therefore, the overhead introduced by conversion is negligible.

Summary: ACORN is efficient in detecting vulnerabilities in multilingual Rust programs with acceptable overhead.

7.6. RQ3: Comparison with Peer Tools

For peer comparison, we used the closest, state-of-the-art baselines we can find: miri [78], MirChecker [79], Rudra [80] and FFIChecker [81]. As shown in the last five columns in TABLE I, the first three Rust program analysis tools can not detect any vulnerabilities caused by cross-language interactions in Rust-C programs, because all of them are based on IRs of Rust, and treat external code as a black-box. Besides that, FFIChecker, a static analysis tool for detecting potential bugs caused by incorrect use of Rust FFI, identified 7 memory vulnerabilities due to the limitations on its analysis algorithms which only focus on the heap memory management issues. ACORN can effectively detected all present vulnerabilities in the micro-benchmark.

Summary: ACORN outperformed the state-of-the-art xtools in performing cross-language program analysis.

7.7. Case Studies

To demonstrate the ACORN's capability of cross-language program analysis intuitively and provide a more concrete understanding of its effectiveness, we present a Double-Free bug detected by ACORN in a Rust-C program and its corresponding Wasm code snippet in Figure 7 for comparison.

As shown in Figure 7, Rust calls a foreign function `c_func`, which is defined in the external C program. Due to the isolation brought by language disparities, Rust is unaware that the `Box` object has been deallocated manually in the C function `c_func` (line c4), and drop it automatically when the `Box` object goes out of scope (line r6), which triggers a Double-Free (DF) bug.

ACORN eliminates the isolation by translating both Rust and C code to Wasm. As we can see in Figure 7, the C function `c_func` (line w4-w8), the Rust function `fn main` (line w10-w26) and other library functions called are all defined in the same translated Wasm `module`. Thus, calling foreign function `c_func` through FFI (line r4) in the original Rust-C program, has been turned into calling local function defined in the same module, which not only eliminates the language disparities, but also removes the boundary imposed by FFI. Then, we performed program analysis on the Wasm code, making it easy to obtain a complete call graphs with a unified trace of memory allocation and release. As a result, ACORN is able to detect that the memory occupied by the Box object was freed twice (line w21 and line w24).

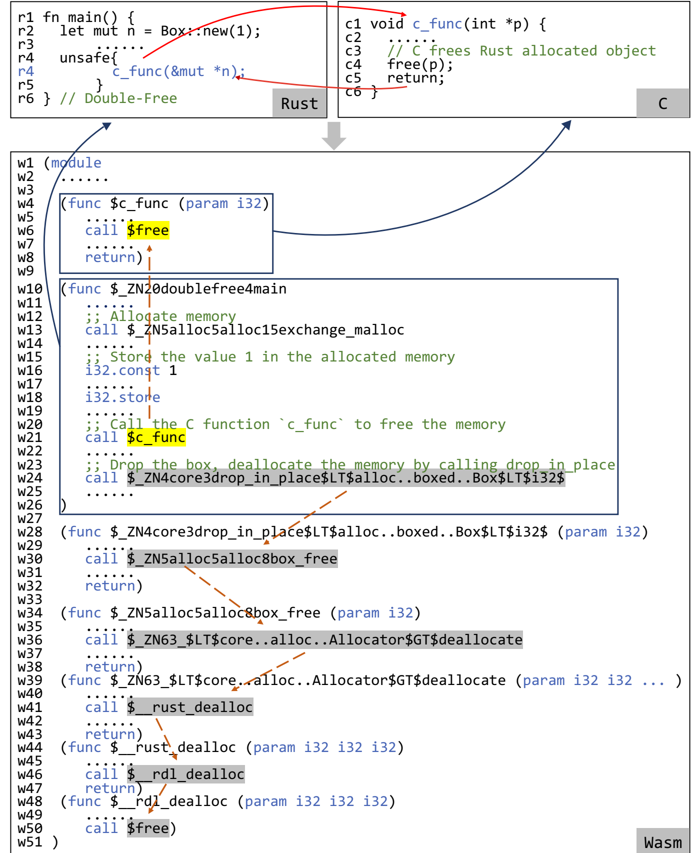

Figure 7: A Double-Free (DF) bug in a multilingual Rust program detected by ACORN.

8. DISCUSSION

In this section, we discuss some possible enhancements to this work, along with directions for future work. It should be noted that this work represents a new step towards defining a holistic and effective static analysis framework for multilingual Rust applications.

More comprehensive types of vulnerabilities. While ACORN demonstrates its effectiveness mainly in identifying memory-related vulnerabilities within multilingual Rust applications, its potential for contribution extends beyond this specific domain. The design of the framework and the potential of Wasm that power ACORN's analysis possess the inherent flexibility to encompass a broader spectrum of vulnerabilities present in software systems. For example, we could extend ACORN to detect concurrency vulnerabilities, by enhancing Wasm's support for signal and develop new algorithms leveraging recent studies in this direction [82] [83]. We leave these important directions for future work.

Supporting other Languages. While we have focused on multilingual Rust programs in this work and experiment results demonstrated its effectiveness, real-world multilingual systems may use a great variety of combinations of different languages. Fortunately, our design facilitates adding support for other languages and new combinations into the current framework.

Specifically, both the specification language and analysis algorithms on it are independent of the conversions. Hence, adding support for other language combinations only requires the addition of new conversions without changing either the target language or the analysis. As a result, our approach can be adapted to analyze other language combinations as well, as long as they support the compilation to Wasm. In the short term, we plan to study multilingual Python programs with ACORN, as PolyCruise [84].

9. Related Work

In recent years, extensive research efforts have been dedicated to Rust security, Wasm security, and the security of multilingual programs. Nevertheless, this study constitutes a novel and distinctive contribution to these domains.

Rust security. In the past few years, there have been a lot of studies on Rust security such as empirical study and vulnerability detection. Current empirical studies on Rust security mainly focus on security vulnerabilities and `unsafe` Rust. Qin et al. [12] conducted an empirical study of memory and concurrency security vulnerabilities in Rust applications. Xu et al. [11] conducted an in-depth study of 186 memory security-related CVEs and proposed a taxonomy. Astrauskas et al. [85] studied the use of *unsafe* in 31867 Rust crates and summarized the usage scenarios of *unsafe*. SafeDrop [14], Rudra [15], Mirchecker [16], and Rupair [17] all perform vulnerability detection based on program analysis. However, the above work is limited to pure Rust code, which cannot analyze multilingual Rust systems, and therefore cannot detect vulnerabilities caused by the interactions between Rust and other languages.

Program analysis for Wasm. Program analysis for Wasm has undergone extensive research, with several notable studies contributing significantly to this domain. Haas et al. [32] introduced an operational semantics and a type system to ensure the safety of Wasm programs. Szanto et al. [34] and Fu et al. [35] conducted taint analysis to trace data propagation for Wasm. Stiévenart et al [33] developed an information flow analysis algorithm, and Lopes et al. [72] presented a vulnerability detection framework utilizing a code property graph. Furthermore, Pradel et al. [73] designed a framework for dynamically analyzing Wasm. In contrast, we introduce a novel concept of utilizing Wasm as a unified specification, enabling us to conduct holistic cross-language program analysis for multilingual Rust applications.

Multilingual application security. Many studies have addressed the security aspects of multilingual applications. Mergendahl et al. [22] introduce the threat model for analyzing cross-language attacks on Rust and Go. Morrisett et al. [86] extend JVML to model the semantics of C to perform inter-Language analysis across Java and C. Li et al. [25] design and implement a pass in LLVM to identify cross-language memory vulnerability in Rust-C/C++ programs. Jiang et al. [84] present PolyCruise, a dynamic analysis framework, for information flow analysis in multilingual systems. Hu et al. [23] formalize the Rust/C programs representable by an intermediate representation and effectively detect memory safety vulnerabilities and integer overflows. Our work differs from the above efforts in that we propose using a completely unified analysis platform, Wasm, which has a promising ability to detect cross-language bugs.

10. Conclusion

This paper presents ACORN, a novel holistic program analysis framework for analyzing multilingual Rust programs. ACORN first utilises Wasm to serve as a unified analysis platform by translating multilingual Rust programs to it, on which cross-language program analysis can be performed. We have implemented a prototype system for ACORN and conducted extensive experiments. Experimental results show that ACORN can effectively detect vulnerabilities across Rust and C by outperforming peer tools. This work represents a new step towards holistic analysis of multilingual Rust programs, making the promise of Rust as a secure system language a reality.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work is partially supported by the National Natural Science Foundation of China (No.62072427, No.12227901), the Project of Stable Support for Youth Team in Basic Research Field, CAS (No.YSBR-005), Academic Leaders Cultivation Program, USTC.

References

[1] "Rust programming language," https://www.rust-lang.org/.

[2] "Ownership," https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html.

[3] "Lifetimes," https://doc.rust-lang.org/rust-by-example/scope/lifetime.html.

[4] W. Bugden and A. Alahmar, "Rust: The programming language for safety and performance," Jun. 2022.

[5] "Tock embedded operating system," https://www.tockos.org/.

[6] S. Lankes, J. Breitbart, and S. Pickartz, "Exploring rust for unikernel development," in *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*, ser. PLOS'19. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 8–15.

[7] M. Sung, P. Olivier, S. Lankes, and B. Ravindran, "Intra-unikernel isolation with intel memory protection keys," in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '20. New York, NY, USA: Association for Computing Machinery, Mar. 2020, pp. 143–156.

[8] "Servo, the parallel browser engine," https://servo.org/.

[9] "Tokio - an asynchronous rust runtime," https://tokio.rs/.

[10] "Unsafe," https://doc.rust-lang.org/std/keyword.unsafe.html.

[11] 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.

[12] 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 Transactions on Software Engineering and Methodology*, vol. 31, no. 1, pp. 3:1–3:25, Sep. 2021.

[13] 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*, Jun. 2020, pp. 246–257.

[14] M. Cui, C. Chen, H. Xu, and Y. Zhou, “Safedrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis,” Apr. 2021.

[15] 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*, Oct. 2021, pp. 84–99.

[16] 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*, Nov. 2021, pp. 2183–2196.

[17] 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*, Dec. 2021, pp. 812–823.

[18] H. M. J. Almohri and D. Evans, “Fidelius charm: Isolating unsafe rust code,” in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, Mar. 2018, pp. 248–255.

[19] 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*, Jun. 2020, pp. 234–245.

[20] E. Rivera, S. Mergendahl, H. Shrobe, H. Okhravi, and N. Burow, “Keeping safe rust safe with galeed,” in *Annual Computer Security Applications Conference*, Dec. 2021, pp. 824–836.

[21] “Ffi,” https://doc.rust-lang.org/nomicon/ffi.html.

[22] S. Mergendahl, N. Burow, and H. Okhravi, “Cross-language attacks,” in *Proceedings 2022 Network and Distributed System Security Symposium*, 2022.

[23] 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.

[24] “Mir,” https://rustc-dev-guide.rust-lang.org/mir/index.html.

[25] Z. Li, J. Wang, M. Sun, and J. C. S. Lui, “Detecting cross-language memory management issues in rust,” *Computer Security – ESORICS 2022*, vol. 13556, pp. 680–700, 2022.

[26] “Llvm language reference manual,” https://llvm.org/docs/LangRef.html.

[27] “Webassembly,” https://webassembly.org/.

[28] “Rust to webassembly,” https://developer.mozilla.org/en-US/docs/WebAssembly/Rust_to_Wasm.

[29] “C/c++ to webassembly,” https://developer.mozilla.org/en-US/docs/WebAssembly/C_to_Wasm.

[30] “Python to webassembly,” https://pythondev.readthedocs.io/wasm.html.

[31] “Javascript to webassembly,” https://github.com/bytecodealliance/javy.

[32] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, “Bringing the web up to speed with webassembly,” in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 2017, pp. 185–200.

[33] Q. Stievenart and C. D. Roover, “Compositional information flow analysis for webassembly programs,” in *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Adelaide, Australia: IEEE, Sep. 2020, pp. 13–24.

[34] A. Szanto, T. Tamm, and A. Pagnoni, “Taint tracking for webassembly,” Jul. 2018.

[35] W. Fu, R. Lin, and D. Inge, “Taintassembly: Taint-based information flow control tracking for webassembly,” Feb. 2018.

[36] “Cwe - cwe-658: Weaknesses in software written in c (4.12),” https://cwe.mitre.org/data/definitions/658.html.

[37] “Internet for people, not profit — mozilla (us),” https://www.mozilla.org/en-US/.

[38] “Types,” https://doc.rust-lang.org/reference/types.html.

[39] J.-Y. Girard, “Linear logic,” *Theoretical Computer Science*, vol. 50, no. 1, pp. 1–101, 1987.

[40] J. Boyland, “Alias burying: Unique variables without destructive reads,” *Software: Practice and Experience*, vol. 31, no. 6, pp. 533–553, May 2001.

[41] D. J. Pearce, “A lightweight formalism for reference lifetimes and borrowing in rust,” *ACM Transactions on Programming Languages and Systems*, vol. 43, no. 1, pp. 1–73, Mar. 2021.

[42] “Smoltcp,” https://github.com/smoltcp-rs/smoltcp, Jul. 2023.

[43] “Tikv/tikv,” TiKV Project, Jul. 2023.

[44] “Blockchain infrastructure for the decentralised web — parity technologies,” https://www.parity.io/.

[45] “A proactive approach to more secure code — msrc blog — microsoft security response center,” https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/.

[46] “Google online security blog: Rust in the android platform,” https://security.googleblog.com/2021/04/rust-in-android-platform.html.

[47] "Rust for linux," https://github.com/Rust-for-Linux.
[48] "Going public launch bug · issue #150 · webassembly/design," https://github.com/WebAssembly/design/issues/150.
[49] "Roadmap-webassembly," https://webassembly.org/roadmap/.
[50] "Webassembly core specification," https://www.w3.org/TR/wasm-core-1/.
[51] "Webassembly becomes a w3c recommendation," https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en.
[52] "Standardizing wasi: A system interface to run webassembly outside the web," https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface.
[53] "Webassembly high-level goals - webassembly," https://webassembly.org/docs/high-level-goals/.
[54] "Security - webassembly," https://webassembly.org/docs/security/.
[55] M. Kim, H. Jang, and Y. Shin, "Avengers, assemble! survey of webassembly security solutions," in *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, Jul. 2022, pp. 543–553.
[56] "wasmcloud," https://wasmcloud.com/.
[57] R. Liu, L. Garcia, and M. Srivastava, "Aerogel: Lightweight access control framework for webassembly-based bare-metal iot devices," in *2021 IEEE/ACM Symposium on Edge Computing (SEC)*, 2021, pp. 94–105.
[58] W. Chen, Z. Sun, H. Wang, X. Luo, H. Cai, and L. Wu, "Wasai: Uncovering vulnerabilities in wasm smart contracts," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Jul. 2022, pp. 703–715.
[59] C. Kelton, A. Balasubramanian, R. Raghavendra, and M. Srivatsa, "Browser-based deep behavioral detection of web cryptomining with coinspy," in *Proceedings 2020 Workshop on Measurements, Attacks, and Defenses for the Web*, 2020.
[60] N. He, R. Zhang, H. Wang, L. Wu, X. Luo, Y. Guo, T. Yu, and X. Jiang, "Eosafe: Security analysis of eosio smart contracts," p. 19.
[61] W. Bian, W. Meng, and Y. Wang, "Poster: Detecting webassembly-based cryptocurrency mining," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, Nov. 2019, pp. 2685–2687.
[62] "Serverless edge compute solutions — fastly," https://www.fastly.com/products/edge-compute.
[63] "Scalar.video - let your creativity run wild on an infinite canvas." https://www.url.ie/a.
[64] "Game boy color emulator library,," https://github.com/torch2424/wasmBoy.
[65] "Language details of the firefox repo," https://4e6.github.io/firefox-lang-stats/.
[66] A. Paszke, S. Gross, F. Massa, and A. Lerer, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, vol. 32, Sep. 2019.
[67] C. R. Harris, K. J. Millman, and S. J. van der Walt, "Array programming with numpy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020.
[68] "Java native interface," https://docs.oracle.com/javase/8/docs/technotes/guides/jni/.
[69] "Python/c api reference manual," https://docs.python.org/3/c-api/index.html.
[70] "Crates.io: Rust package registry," https://crates.io/.
[71] "Webassembly · golang/go wiki," https://github.com/golang/go/wiki/WebAssembly.
[72] T. Brito, P. Lopes, N. Santos, and J. F. Santos, "Wasmati: An efficient static vulnerability scanner for webassembly," *Computers & Security*, vol. 118, p. 102745, Jul. 2022.
[73] D. Lehmann and M. Pradel, "Wasabi: A framework for dynamically analyzing webassembly," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Apr. 2019, pp. 1045–1058.
[74] "Move - rust," https://doc.rust-lang.org/std/keyword.move.html.
[75] "Copy in std::marker - rust," https://doc.rust-lang.org/std/marker/trait.Copy.html.
[76] "Wasmati: A generic and efficient code property graph infrastructure forscanning vulnerabilities in webassembly code," https://github.com/wasmati/wasmati.
[77] "Wasabi: A dynamic analysis framework for webassembly programs." https://github.com/danleh/wasabi.
[78] "Rust-lang/miri: An interpreter for rust's mid-level intermediate representation," https://github.com/rust-lang/miri.
[79] "Rust-mir-checker," https://github.com/lizhuohua/rust-mir-checker.
[80] "Rudra: Rust memory safety & undefined behavior detection," https://github.com/sslab-gatech/Rudra.
[81] "Rust-ffi-checker," https://github.com/lizhuohua/rust-ffi-checker.
[82] C. Watt, A. Rossberg, and J. Pichon-Pharabod, "Weakening webassembly," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–28, Oct. 2019.
[83] P. Ning and B. Qin, "Stuck-me-not: A deadlock detector on blockchain software in rust," *Procedia Computer Science*, vol. 177, pp. 599–604, 2020.
[84] W. Li, J. Ming, X. Luo, and H. Cai, "Polycruise: A cross-language dynamic information flow analysis," in *31st USENIX Security Symposium*, 2022, pp. 2513–2530.
[85] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging rust types for modular specification and verification," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–30, Oct. 2019.
[86] G. Tan and G. Morrisett, "Ilea: Inter-language analysis across java and c," *ACM SIGPLAN Notices*, vol. 42, no. 10, pp. 39–56, Oct. 2007.