ACORN: Towards a Unified Cross-Language Program Analysis Framework for Rust

Lei Xia, and Baojian Hua*
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*
* Corresponding authors.

Abstract—Rust is an emerging programming language enforcing safety guarantees by novel language features and strict compile-time checking rules, which has been used extensively to build system software. In practice, multilingual Rust applications containing external C code, memory security vulnerabilities can occur due to the intrinsically unsafe nature of C and the improper interactions between Rust and C. Unfortunately, most existing security studies on Rust only focus on pure Rust code but cannot analyze either the native C code or the interactions between Rust and C in multilingual Rust applications. As a result, the lack of such studies may defeat the guarantee that Rust is a safe language.

This paper presents ACORN, a unified program analysis framework that spans Rust and C. ACORN enables program analyses to comprehend the semantics of C code by translating Rust and C into a unified specification language. The ACORN framework comprises three key components: (1) ACRONIR, a source-neutral low level language serving as the unified specification language for program analysis; (2) a transformation mechanism that constructs models by converting Rust and C code into ACRONIR; and (3) program analysis algorithms on ACRONIR to detect security vulnerabilities. We have implemented a software prototype for ACORN and conducted extensive experiments to evaluate its effectiveness and performance. The experimental results demonstrate that ACORN effectively detects common memory security vulnerabilities caused by the interaction between Rust and C that are overlooked by state-of-the-art tools.

Keywords–Rust, Security, Multilingual Program Analysis

1. INTRODUCTION

Rust [1] is an emerging systems-level programming language designed to provide both strong security guarantees and high performance. Specifically, Rust has introduced a set of unique language features that emphasize security. Among these language features, the ownership and borrowing system is considered the cornerstone of Rust's memory safety [2]. Additionally, combined with automatic lifetime-based memory management [3] and strict security checking rules, Rust effectively mitigates prevalent memory security vulnerabilities such as use-after-free, double free and buffer overflow [4]. Furthermore, Rust upholds its performance comparable to that of C/C++ by embracing a zero-cost abstraction philosophy [5]. One way Rust achieves this is by incorporating the programming concept of explicit lifetime [3], which incurs no runtime overhead without using garbage collections [6].

Although it is widely believed that gradually reimplementing security-critical components of software in Rust enhances software security, there are potential security threats when calling external code written in other languages through the Foreign Function Interface (FFI) supported by Rust [7]. On one hand, the Rust compiler cannot perform security checks across the FFI boundaries [8]. On the other hand, other languages do not enforce Rust's rules and guarantees. This lack of enforcement enables attackers to navigate between the FFI boundaries and exploit these vulnerabilities [9] . Recent empirical studies [4] [10] have shown that the incorrect use of FFI is one of the primary causes of real world memory safety bugs.

Unfortunately, most of existing security studies on Rust [11] [12] [13] [14] only focus on Rust it self, ignoring the security threats which may be raised by the interactions between Rust and external code written in other languages via FFI. Although some studies [15] [16] have conducted cross-language programming analysis on some unified specification language by translating both Rust and other languages such as C/C++ to them, these unified specification languages based on traditional IRs such as LLVM IR [17] and MIR [18] serve whole-program translation fully covering the original code semantics, which is too heavyweight and laborious for various languages when targeting a unified IR.

We propose that a comprehensive and standardized specification language is necessary for removing boundaries between Rust and other languages in multilingual Rust programs and conducting cross-language analysis. To satisfy these requirements, the specification language should possess the following qualities: 1) *Extensibility*: the language must have the capability to represent a wide range of programming languages through effortless conversions. This enables the analysis of Rust's interaction with different languages; 2) *Neutrality*: it should remain impartial towards source code, ensuring unbiased program analysis; 3) *Security*: Strong security mechanisms must be embedded in the language to ensure the reliability and integrity of the analysis; and 4) *Tooling and Analysis Support*: the specification language should ideally have a robust ecosystem of analysis tools and frameworks that can be readily reused and ported for cross-language analysis. To this end, the approach that we propose is to define the specification language, dubbed as ACRONIR, based on WebAssembly (Wasm), for cross-language program analysis in multilingual Rust programs. First, numerous programming languages, such as Rust [19], C/C++ [20], Go [21], Python [22] and JavaScript [23], have official support for compiling to WebAssembly (Wasm). This support enables the analysis and verification of multilingual Rust programs across different language boundaries. While originally devised for execution purposes [24], the features of Wasm make it exceptionally well-suited as a platform for cross-language analysis. Next, binary format of WebAssembly (Wasm) is intentionally designed to be independent of the source language [24], offering the opportunity for unbiased analysis and enabling consistent treatment of multilingual Rust programs. Furthermore, Wasm incorporates inherent safety and security features such as type safety mechanisms, sandboxing isolation and control flow integrity [25] [26]. Lastly, Program analysis on Wasm has been extensively researched and several notable studies have contributed to this field [26] [27] [28] [29] [30] , which provide valuable insights and techniques that can be leveraged for cross-language analysis based on Wasm.

In this paper, we propose ACORN, a unified cross-language program analysis framework for multilingual Rust programs. This framework enables existing Wasm program analyses to model and understand the semantics of both Rust and other languages supporting the compilation to Wasm in multilingual Rust applications, by simultaneously translating them to Wasm. The ACORN framework consists of three main components: (1) A unified and formal specification language, called ACRONIR, is proposed as a source-independent, low-level intermediate representation with a strong type system and security measures, aiming to enhance the precision and reliability of the analysis significantly; (2) a transformation from multilingual Rust programs to ACRONIR; and (3) program analysis algorithms on ACRONIR to detect security vulnerabilities.

Such a design brings two significant advantages to our unified program analysis framework: (1)Extensibility and (2) simplicity; Firstly, extensibility of ACORN exhibits good extensibility for both the programming languages being analyzed and the program analysis algorithms. It allows for the extension of languages that support compilation to WebAssembly (Wasm) and enables the development of new program analysis algorithms. Additionally, most existing Wasm program analysis algorithms can be seamlessly integrated into ACORN with minimal or no modifications. Secondly, the design of ACORN simplifies its implementation by leveraging existing translation tools and Wasm program analysis algorithms.

To validate our design, we implemented ACORN for Rust-C, as many C projects integrate Rust into their existing codebases, such as the Linux Kernel, to enhance security. We conducted experiments to evaluate the effectiveness and performance of our implementation. First, we assessed the effectiveness of ACORN by testing it on two datasets: 1) a micro-benchmark containing common memory security vulnerabilities resulting from interactions between Rust and C, and 2) real-world vulnerability sets []. The experimental results demonstrated that ACORN can effectively detect these vulnerabilities. Furthermore, our evaluation shows that ACORN is efficient, incurring an average runtime overhead of 0.26 seconds.

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

- **A unified program analysis framework that works across Rust and C.** We present ACORN, a program analysis framework working across the language boundaries in multilingual Rust programs by translating them to ACRONIR, serving as a unified specification language.
- **A prototype implementation of ACORN for Rust-C programs.** We implemented a prototype of ACORN, by translating Rust and C code into ACRONIR, and by porting existing Wasm program analysis to ACRONIR to analysis multilingual program written in Rust and C.
- **Evaluation of ACORN.** We conducted extensive experiments to evaluate the effectiveness and performance of ACORN. Experimental results demonstrated that ACORN can effectively detect vulnerabilities across Rust and C with minimal additional overhead.

Outline. The rest of this paper is organized as follows. Section 2 introduces the background knowledge for this work. Section 3 presents the motivation for this work. Section 4 presents a formal definition of the syntax of ACRONIR, and formally describes the translation rules from Rust and C to ACRONIR. Section 6 presents a prototype implementation. Section 7 presents the evaluations we conducted. Section 8 discusses limitations and future work. Section 9 describes related work, and Section 10 concludes.

2. BACKGROUND

To be self-contained, this section presents the necessary background knowledge on Rust (Section 2-A), Wasm (Section 2-B), and multilingual program analysis (Section 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 [31]. 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 [32] based on linear logic [33] and alias types [34]. 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 [5]. Second, Rust achieve high efficiency through the ownership-based explicit memory management and a lifetime model, without any garbage collectors [6]. 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 [35] [36] [37], Web browsers [38], file systems [39], network protocol stacks [40], language runtime [41], databases [42], and blockchains [43]. Rust has also been a favored option for game development, as exemplified by games such as Oxide [44] and Second Life [45]. Moreover, Rust is gaining more adoptions in the industry, such as Microsoft [46], Google [47], and even Linux [48] are beginning to use Rust for the development of system software.

2.2. Wasm

Brief history. Wasm was introduced by Google and Mozilla in 2015 [49] and quickly gained popularity, becoming a de facto standard language in browsers by 2017 [50]. In 2018, the first complete formal definition of Wasm was released [51], solidifying its specifications. The W3C officially recognized Wasm as the fourth Web standard in 2019 [52]. Over time, Wasm has evolved and matured, with the development of the WebAssembly System Interface (WASI) [53] and the ongoing work on the standard version 2.0 draft [51]. 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 [24]. First, to ensure program safety, Wasm incorporates secure features such as strong typing, sandboxing isolation and control flow integrity [54] [25]. 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 Web domains, 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. Developers can take advantage of the strengths of different languages, or using existing libraries or code written in different languages to facilitate programming and improve software performance, hence, it is widely used in many software systems such as Mozzila [65], PyTorch [66] and NumPy [67]. Since multilingual programming put high demand on seamlessly interoperation between different languages, foreign function interface (FFI), a mechanism was proposed to connect different languages. Different languages support FFI in various ways. For example, Java supports runtime environment FFI, *i.e.*, Java Native Interface (JNI) [68], and Rust and Python support FFI [69] [70] on the language level.

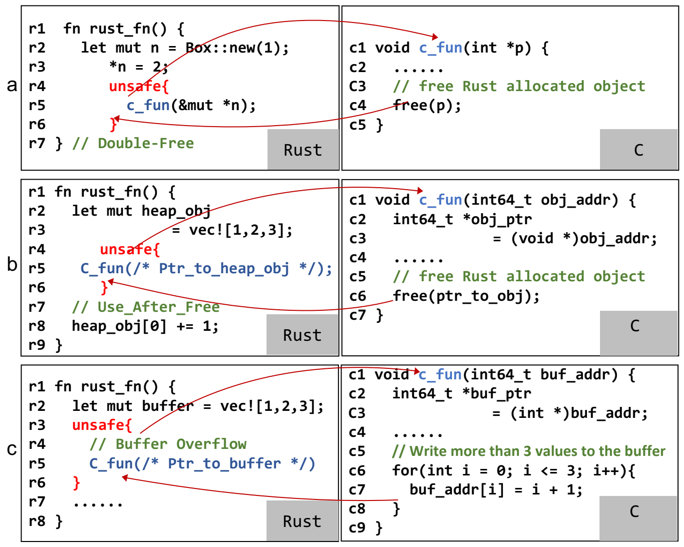
3. Motivation

Rust supports interacting with external code written in other languages, through the foreign function interface (FFI) [7], which is a crucial feature to Rust and brings two advantages to the development of Rust: First, it fosters interoperability of Rust, enabling developers to choose the best tools and languages for specific tasks in Rust applications. For example, in performance-critical scenarios, developers may choose to write low-level code in C or C++ and interface it with Rust; and second, it enables Rust to leverage the benefit of existing numerous mature libraries and codebases, particularly those written in C and C++, without rewriting them entirely, which saves large time and effort.

However, along with the benefits, collaborating with other languages through FFI also brings potential security threats to multilingual Rust programs. On the one hand, the Rust compiler cannot perform unified security checks on Rust code and external code across the FFI boundaries, thus, FFI is a kind of `unsafe` Rust [8]; on the other hand, external code written in other languages, do not enforce language rules and security guarantees of Rust, which can lead to some conflicts, such as in the memory management. To better illustrate the security risks associated with the cross-language interaction through FFI, Figure 1 demonstrates three common memory vulnerabilities in multilingual Rust programs.

Double-Free. A double-free is a serious memory bug which might lead to various issues, such as crashes, unpredictable behavior, and security vulnerabilities. Figure 1(a) presents a Double-Free(DF) due to interactions between Rust and C. First, Rust defines a variable `n`, and its value is a `Box` pointing to the value 5 assigned on the heap (line r2 of Figure 1(a)). In Rust, the Box object will be automatically deallocated when it goes out of scope, as `n` does at the end of `rust_fn` (line r7 of Figure 1(a)), which means that Rust will free the memory occupied by the integer pointed to by `n`. Then, the Rust code calls the C function (`c_fun`) with the mutable pointer to `n` (line r5 of Figure 1(a)), in which, the Box object is deallocated manually. When returning from the C function, the Box object will be automatically deallocated when it goes out of scope, as `n` does at the end of `rust_fn` (line r7 of Figure 1(a)), which means that Rust will again free the memory occupied by the integer pointed to by `n` and trigger a Double-Free (DF) vulnerability.

Use-After-Free. A use-after-free occurs when a program use a memory cell after it has been deallocated. Figure 1(b) illustrates a Use-after-Free (UaF) vulnerability in multilingual Rust programs. First, in the Rust function `rust_fun`, a heap object `heap_obj` is allocated and initialized (line r2-r3 of



```
r1  fn rust_fn() {
r2    let mut n = Box::new(1);
r3       *n = 2;
r4       unsafe{
r5         c_fun(&mut *n);
r6       }
r7 } // Double-Free
```

```
c1 void c_fun(int *p) {
c2   ......
C3   // free Rust allocated object
c4   free(p);
c5 }
```

```
r1 fn rust_fn() {
r2   let mut heap_obj
r3                 = vec![1,2,3];
r4      unsafe{
r5    C_fun(/* Ptr_to_heap_obj */);
r6      }
r7   // Use_After_Free
r8   heap_obj[0] += 1;
r9 }
```

```
c1 void c_fun(int64_t obj_addr) {
c2   int64_t *obj_ptr
c3              = (void *)obj_addr;
c4   ......
c5   // free Rust allocated object
c6   free(ptr_to_obj);
c7 }
```

```
r1 fn rust_fn() {
r2   let mut buffer = vec![1,2,3];
r3   unsafe{
r4     // Buffer Overflow
r5     C_fun(/* Ptr_to_buffer */)
r6   }
r7   ......
r8 }
```

```
c1 void c_fun(int64_t buf_addr) {
c2   int64_t *buf_ptr
C3              = (int *)buf_addr;
c4   ......
c5   // Write more than 3 values to the buffer
c6   for(int i = 0; i <= 3; i++){
c7     buf_addr[i] = i + 1;
c8   }
c9 }
```

Figure 1: Sample code illustrating three kinds of memory vulnerabilities across Rust and C.

Figure 1(b)), then the pointer to the heap object is passed to the external C function `c_fun` (line r5 of Figure 1(b)). The external C function deallocated the memory saving the value of `heap_obj` (line c6 of Figure 1(b)). After calling the C function, the Rust code (line r8 of Figure 1(b)), when attempts to access and modify the elements of the `heap_obj` which has been previously deallocated in the C function, triggers a Use-after-Free (UaF) vulnerability.

Buffer Overflow means that a program writes more data into a buffer (such as a contiguous block of memory) than it can hold, as the Buffer Overflow bug presented in Figure 1(c). A mutable `Vector` of length tree is created and initialized in Rust code (line r2 of Figure 1(c)), and then, the pointer to the `buffer` is passed as an argument to the C function (line r5 of Figure 1(c)) , in which, a loop is used to write more than 3 values to the `buffer`. However, the `buffer` in Rust is allocated to hold only 3 values, which results in a Buffer Overflow. This extra data overflows into adjacent memory locations, potentially overwriting important information, corrupting data, and causing unintended consequences.

Unfortunately, calling external function is one of the most common `unsafe` operations in Rust, accounting for 22.5% of all `unsafe` function calls [10] and more than 72% of packages on the official Rust package registry (crates.io [71]) depending on at least one unsafe FFI-bindings package [16].Recent empirical studies [4] [10] have shown that the incorrect use of FFI is one of the primary causes of real world memory safety bugs.

4. A Unified Specification Language

The key insight of our proposed framework is to construct a specification suitable for both C/C++ and Rust, then make this specification comprehensible to existing analysis tools. Given this insight, two main questions remain: (1) "What specification language would be the best choice for leveraging existing research and tools for program analysis ?" and (2) "How to generate such specifications for Rust and C/C++ code?"

This section will answer the first question, by presenting the advantages of using WebAssembly (Wasm) as the specification language (Section 4-B), and the syntax of AcronIR (Section 4-C). Then, in Section (Section 5), we will answer the second question by presenting the conversion rules from C to AcronIR.

4.1. Specification Language Design Rationale

Generality and Compatibility. The specification language should be both general and compatible. It should support multiple programming languages without favoring any particular one, providing a neutral ground for faithful representation and analysis of code from different languages. This ensures unbiased analysis and allows for consistent treatment of multilingual programs. Additionally, the specification language must be compatible with the source languages being analyzed, enabling seamless translation of code and accurate representation of its behavior. This compatibility facilitates precise cross language analysis, ensuring that analysis results accurately reflect the intended behavior of the multilingual program.

Formality and Precision. The specification language must possess well-defined and unambiguous semantics. It should offer a clear and unequivocal interpretation of program behavior, eliminating any potential ambiguity or room for subjective interpretation. This precision in semantics guarantees accurate analysis results, ensuring the reliability and reproducibility of the analysis across various tools and environments.

Expressiveness. The specification language ought to possess sufficient expressiveness to accurately capture the semantics and behavior of diverse programming languages. It should encompass constructs that enable the representation of various language features, control flow patterns, data structures, and type systems. Additionally, the language should facilitate precise modeling of program behavior, thereby enabling comprehensive analysis spanning different language boundaries.

Modularity and Compositionality. The specification language ought to facilitate modular and compositional analysis by supporting the decomposition of programs into smaller, manageable units. This capability enables modular analysis and fosters compositional reasoning, providing a framework for code reuse and reducing analysis complexity. Moreover, the language's modularity facilitates scalability, allowing analysis techniques to be applied effectively to large multilingual codebases.

Tooling and Analysis Support. The specification language should possess a robust ecosystem of tools and analysis frameworks. It should encompass a diverse array of analysis tools, libraries, and frameworks that harness the specification language for a wide range of program analysis tasks. These tools should encompass static analysis, dynamic analysis, testing, verification, and security analysis, offering developers a comprehensive toolkit for proficient multilingual program analysis.

4.2. Advantages of Wasm

WebAssembly (Wasm) serves as an advantageous choice for a source-neutral low-level language when designing a specification language for cross-language program static analysis. This selection offers a range of benefits and opportunities for efficient and effective analysis across diverse programming languages. The advantages of utilizing Wasm in this context can be summarized as follows.

First and foremost, Wasm's source-neutrality facilitates seamless integration and analysis of code written in various programming languages. By adopting Wasm as the specification language, it becomes possible to analyze and verify programs across different language boundaries, thereby enabling a unified approach to cross language static analysis. This versatility proves particularly valuable in heterogeneous codebases where multiple programming languages coexist.

Additionally, Wasm's well-defined and compact bytecode representation enhances the feasibility and accuracy of static analysis. Wasm's bytecode is designed for efficient execution, and its structured nature simplifies program analysis tasks, including control flow analysis, data flow analysis, and type analysis. Leveraging Wasm's bytecode representation as the specification language provides a solid foundation for precise and comprehensive static analysis across multiple languages.

Moreover, Wasm incorporates inherent safety and security features that contribute to the reliability of static analysis. By design, Wasm enforces a sandboxed execution environment and provides memory isolation, mitigating certain classes of security vulnerabilities. Utilizing Wasm as the specification language empowers static analysis tools to leverage these built-in security features, enabling the detection and prevention of potential security issues across different programming languages.

Furthermore, Wasm's performance optimizations optimize the efficiency of static analysis processes. The low-level nature of Wasm's bytecode representation allows static analysis tools to operate on a compact and efficient program representation. This not only expedites the analysis itself but also minimizes resource consumption, ultimately improving the scalability and responsiveness of cross-language static analysis workflows. In conjunction with its advantages for static analysis, Wasm benefits from a rich ecosystem of tooling and frameworks specifically tailored for Wasm-based analysis tasks. This existing support provides a valuable resource for developers, offering functionalities such as program slicing, reachability analysis, and vulnerability detection. By adopting Wasm as the specification language, one can readily leverage these established tools and frameworks, accelerating the development and deployment of effective cross-language static analysis solutions.

In summary, the utilization of WebAssembly (Wasm) as a source neutral low-level language for the specification language in cross language program static analysis brings several advantages. Wasm's source-neutrality enables unified analysis across multiple programming languages, its compact bytecode representation simplifies analysis tasks, its inherent safety features enhance security, its performance optimizations improve efficiency, and its established tooling ecosystem provides valuable support. These advantages collectively contribute to more accurate, scalable, and reliable static analysis across diverse codebases, fostering the development of secure and robust software systems.

$$
\begin{array}{llll}
\text{Val. Type} & \rho & ::= & \texttt{i32} \mid \texttt{i64} \mid \texttt{f32} \mid \texttt{f64} \\
\text{Func. Type} & \sigma & ::= & \rho^* \to \rho^* \\
\text{Type} & \tau & ::= & \rho \mid \sigma \\
\text{Binary Op.} & b & ::= & \texttt{i32.add} \mid \texttt{i32.mul} \mid \texttt{i32.shl} \mid \ldots \\
\text{Unary Op.} & u & ::= & \texttt{i32.abs} \mid \texttt{i32.eqz} \mid \ldots \\
\text{Load/Store} & l & ::= & \rho.\texttt{load} \mid \rho.\texttt{store} \\
\text{Local Op.} & c & ::= & \texttt{local}.(\texttt{set} \mid \texttt{get} \mid \texttt{tee})\ x \\
\text{Global Op.} & g & ::= & \texttt{global}.(\texttt{set} \mid \texttt{get})\ x \\
\text{Call} & t & ::= & \texttt{call}\ f \mid \texttt{call\_indirect}\ \sigma \\
\text{Instr.} & i & ::= & b \mid u \mid l \mid c \mid g \mid t \\
 & & \mid & \texttt{drop} \mid \texttt{nop} \mid \texttt{if} \mid \texttt{else} \mid \texttt{block} \\
 & & \mid & \texttt{loop} \mid \texttt{end} \mid \texttt{br}\ a \mid \texttt{br\_if}\ a \\
 & & \mid & \texttt{br\_table}\ a^+ \mid \texttt{select} \\
 & & \mid & \texttt{memory.grow} \mid \rho.\texttt{const}\ c \mid \ldots \\
\text{Function} & f & := & \sigma\ x\{i^*\} \\
\text{Module} & m & := & f^*
\end{array}
$$

Figure 2: Core syntax of AcronIR language.

4.3. Syntax

We introduce a simple language model capturing the core syntax of Wasm, to present the syntax for the ACORNIR specification language, as summarized in Fig. 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, memory load/stores, structured control flows, and function invocation/return, with some irrelevant instructions 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 stack and pushes the result. Second, Wasm instructions are strongly typed in specifying the expected type in the opcode (*e.g.*, the `i32` in `i32.abs`), facilitating binary-level type checking. Third, Wasm supports *structured* control flows (*e.g.*, `if` or `loop`), making compilation to Wasm easier.

5. Translation Models

This section presents translation from C/C++ to AcronIR and from Rust to AcronIR, by formally defining compilation rules.

$$
\begin{array}{llll}
\text{Constant} & c & & \\
\text{Bid} & b & \in & \mathbb{N} \\
\text{Variable} & x & \in & \{x_0, x_1, x_2, \ldots\} \\
\text{Type} & t & ::= & \texttt{bool} \mid \texttt{int} \mid \texttt{unit} \mid \ldots \\
\text{Operand} & o & ::= & \texttt{const}\ c \mid \texttt{move}\ x \mid \texttt{copy}\ x \\
\text{Expression} & e & ::= & n \mid x \mid o \mid e + e \\
\text{Statement} & s & ::= & x = e \mid s_1; s_2 \mid \texttt{if}(e)\ s_1 s_2 \\
 & & \mid & \texttt{while}(e)\ s \mid \texttt{match}(e)\{\overrightarrow{n : b}\} \\
 & & \mid & \texttt{storageLive}(x) \\
 & & \mid & \texttt{storageDead}(x) \\
\text{Terminator} & T & ::= & f(x) \mid \texttt{drop}(x) \mid \texttt{break} \mid \texttt{continue} \\
 & & \mid & \texttt{return} \mid \texttt{return}(e) \\
\text{Function} & f & ::= & \texttt{fn}\ \mathit{Fun}(\overrightarrow{x : t}) \rightarrow t\ \{\overrightarrow{b : B}\} \\
\text{Block} & B & ::= & \vec{s}\ T
\end{array}
$$

Figure 3: Core syntax of the Rust language.

$$
\frac{\Phi = \text{mapVar}([t_1\ x_1; \ldots; t_n\ x_n; t'_1\ y_1; \ldots; t'_m\ y_m]) \qquad \tau_{\text{ret}} = \text{mapTy}(t) \qquad \Phi, \tau_{\text{ret}} \vdash s \rightsquigarrow \vec{I}}{\vdash \left(\begin{array}{c} t\ \mathit{Fun-Name}(t_1\ x_1, \ldots, t_n\ x_n) \\ \{t'_1\ y_1; \ldots; t'_m\ y_m; s;\} \end{array} \right) \rightsquigarrow \vec{I}}
$$

Figure 4: Rule for Compiling Functions.

5.1. Formalizing Rust

Figure 3 presents the syntax for a subset of Rust, which is used to specify the rules for translation to ACORN. As our goal is to present the translation rules rigorously, we have included, in this syntax, key features of Rust, such as expressions, statements, and functions. To simplify the presentation, we have omitted some other features such as aggregate types, unions, and so on. However, these features can be added without any technical difficulty.

5.2. Compiling Rust to ACRONIR

The rules for compiling Rust to ACORN are specified by a set of judgments. We use the notation $\vec{I}$ for a sequence of ACRONIR instructions, and $\vec{e}$ for a sequence of expressions. We use the notation $[\,]$ for an empty sequence, and @ for concatenation of instructions, for example, $\vec{I_1} @ \vec{I_2}$ denotes the concatenation of $\vec{I_1}$ and $\vec{I_2}$.

Compiling Rust functions. Figure 4 presents the rules for compiling a Rust function into a sequence of ACRONIR instructions $\vec{I}$. The compiler first constructs a compilation environment Φ. This construction is formalized using two auxiliary functions mapTy(-) and mapVar([-]). The function mapTy(t) maps a Rust type t to a ACRONIR type τ:

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

The function mapVar($[x_1 : t_1; \ldots; x_n : t_n;]$) maps Rust variable declarations to ACRONIR declarations:

$$
\text{mapVar}([x_1 : t_1; \ldots]) = [\text{mapTy}(t_1)\ x_1, \ldots]
$$

The Rust function body s is then compiled under the environment Φ, and a return type τ_{ret}, which is mapped from type t, the return type of the C function.

Compiling expressions. Figure 5 presents rules for compiling expressions, which are formalized by the following judgment:

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

An expression e is compiled into a list of ACRONIR instructions that put the value of the expression on the top of the ACRONIR stack, and also returns the ACRONIR type of the value. An example is the rule for a constant n, in which it just pushes the constant onto the stack.

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

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

$$
\frac{\Phi \vdash e_1 \rightsquigarrow (\vec{I_1}, \texttt{i32}) \qquad \Phi \vdash \texttt{e}_2 \rightsquigarrow (\vec{\texttt{I}_2}, \texttt{i32})}{\Phi \vdash e_1 + e_2 \rightsquigarrow (\vec{I_1} @ \vec{I_2} @ [i32.add], \texttt{i32})}
$$

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

Figure 5: Rules for Compiling Expressions.

Compiling statements. Figure 6 presents rules for compiling C statements, using the judgment

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

A C statement s is compiled to a sequence of ACRONIR instructions.

Most rules are straightforward. As an example, to compile "$\texttt{if}(e)\ s_1 s_2$", we first compile the conditional expression e, then the statement s_1 followed by the statement s_2; then we generate four basic blocks bb0 to bb3, by inserting appropriate comparisons, labels, and jumps. The generated code $\vec{s'_1}$ and $\vec{s'_2}$ from statements s_1 and s_2 are placed at the start of the blocks bb1 and bb2, respectively. Similarly, to compile "$\texttt{while}(e)\ s$", after the statement s has been compiled, the control will jump back to block BB0.

6. PROTOTYPE IMPLEMENTATION

To conduct the evaluation, we have implemented a prototype for ACORN, which consists of two main components: (1) a converter for translating both Rust and C code to ACRONIR; and (2) a portion of existing program analysis algorithms and

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

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

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

$$\frac{\begin{array}{c}\Phi \vdash e \rightsquigarrow (\vec{I_e}, \tau) \\ \Phi \vdash s_1 \rightsquigarrow \vec{I_1} \quad \Phi \vdash s_2 \rightsquigarrow \vec{I_2}\end{array}}{\begin{array}{ll} & \vec{I_e}@[br\_if; label\ else]@ \\ \Phi \vdash \texttt{if}(e)\ s_1\ s_2 \rightsquigarrow & \vec{I_1}@[return]@ \\ & [label\ else]@\vec{I_2}\end{array}}$$

$$\frac{\Phi \vdash e \rightsquigarrow (\vec{I_e}, \tau) \quad \Phi \vdash s \rightsquigarrow \vec{I_s}}{\begin{array}{ll} & \vec{I_e}@[label\ loop]@ \\ \Phi \vdash \texttt{while}(e)\ s \rightsquigarrow & [br\_if; label\ end]@\vec{I_s}@ \\ & [br; label\ loop]@ \\ & [label\ end]\end{array}}$$

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

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

Figure 6: Rules for Compiling Statements.

tools to AcronIR. To implement the converter, we leveraged the compilation to Wasm officially supported by Rust and C. For program analysis comparison, we ported existing algorithms and tools so that they can process AcronIR for cross-language program analysis. To be specific, we ported two state-of-the-art analysis tools: 1) WASMATI [27], a generic and efficient Code property graph infrastructure for scanning vulnerabilities in WebAssembly Code; and 2) WAMDYPA [], a fully automated bug detection framework for Wasm programs based on dynamic analysis in conjunction with static instrumentation, leveraging runtime information that is hard to obtain in static analysis

7. Evaluation

In this section, we conduct experiments to evaluate Acorn. We first propose the research questions to investigate through the experiments (Section 7-A), then, we present the datasets used for the evaluation (Section 7-C). and the experimental results regarding Acorn's effectiveness (Section 7-D), performance (Section 7-E) and the comparison with peer tools (Section 7-F). Finally, we present the case study involving the security vulnerabilities detected by Acorn (Section 7-G).

7.1. Research Questions

To evaluate Acorn and present the experimental results, we mainly investigate the following research questions:
RQ1: Effectiveness. Acorn is designed to be a unified cross-language program analysis framework for Rust, is it effective in detecting potential security vulnerabilities in multilingual Rust programs?
RQ2: Performance. Acorn needs to convert multilingual Rust programs to AcronIR before cross-language analysis on it, what is the overall performance of Acorn?
RQ3: Comparison. Some studies have contributed to this field, how does Acorn compare to these peer 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 used two datasets to conduct the evaluation: 1) micro-benchmarks; and 2) real-world CWE, containing a total of 117 test cases.
Micro-benchmark. We manually constructed a micro-benchmark consisting of 20 test cases with diverse types of vulnerabilities (as presented by the second column of TABLE I), including Buffer Overflow (BO), Use-after-Free (UaF), Double Free (DF) and so on. These test cases are collected from two sources (third column of TABLE I): 1) public CVEs; and 2) existing literature on Rust security studies. To better reflect the essence of these vulnerabilities and to simplify the validation, we have rewritten some of the original buggy code by removing irrelevant code.
Real-world CWE. CWE [72] is a set of widely used "Weaknesses in Software Written in C", with a total of 117 vulnerable programs. Evaluating Acorn on well-established vulnerability sets like CWE demonstrates the effectiveness of Acorn on real-world multilingual applications. To use CWE for the evaluation of Acorn, we added a Rust wrapper to each code example in CWE, turning them into multilingual Rust applications with C.

7.4. Effectiveness of Acorn

To evaluate the effectiveness of Acorn, we first apply Acorn to micro-benchmarks. The last five columns in TABLE I present experimental results, which demonstrate that Acorn is superior to the other four state-of-the-art studies (tools) in that Acorn successfully detected all vulnerabilities in these benchmarks whereas the other four tools detected none. This experiment shows that Acorn is effective in detecting vulnerabilities in multilingual Rust applications.
To investigate the effectiveness of Acorn on real-world programs, we applied Acorn to the CWE, our second benchmark. As TABLE II shows, the CWE dataset contains a total of 117 test cases, among which 48 were filtered out because their compilations failed for two reasons: 1) they contain functions, such as `gets`, that have been removed since the

TABLE I: Experimental results on micro-benchmarks.

Test Case	Vulnerability Type	Source	ACRONIR LOC	CT(s) / pl(ms)	AT(s) / pl(ms)	rustc	Miri	MirChecker	Rudra	ACORN (This work)
1	OOB	[9]	678	0.23 / 2.60	0.46 / 5.33	✗	✗	✗	✗	✓
2	stack overflow	[9]	559	0.22 / 0.40	0.56 / 1.00	✗	✗	✗	✗	✓
3	CFI violation	[9]	750	0.22 / 0.29	0.50 / 0.67	✗	✗	✗	✗	✓
4	meta data leaking	[9]	199	0.23 / 1.17	0.48 / 2.42	✗	✗	✗	✗	✓
5	UaF/DF	[9]	138	0.24 / 1.71	0.46 / 3.35	✗	✗	✗	✗	✓
6	Division by Zero	[16]	134	0.21 / 1.56	0.52 / 3.84	✗	✗	✗	✗	✓
7	OOB	[16]	162	0.23 / 1.39	0.43 / 2.68	✗	✗	✗	✗	✓
8	OOB	[16]	223	0.21 / 0.94	0.42 / 1.90	✗	✗	✗	✗	✓
9	UaF/DF	[15]	309	0.24 / 0.76	0.48 / 1.56	✗	✗	✗	✗	✓
10	integer overflow	[15]	215	0.21 / 0.99	0.54 / 2.52	✗	✗	✗	✗	✓
11	UaF/DF	[15]	356	0.24 / 0.68	0.49 / 1.37	✗	✗	✗	✗	✓
12	UaF/DF	[15]	351	0.23 / 0.64	0.49 / 1.41	✗	✗	✗	✗	✓
13	format string attack	[15]	92	0.23 / 2.47	0.47 / 5.06	✗	✗	✗	✗	✓
14	buffer overflow	[15]	287	0.23 / 0.82	0.54 / 1.88	✗	✗	✗	✗	✓
15	format string attack	[15]	168	0.23 / 2.47	0.47 / 5.06	✗	✗	✗	✗	✓
16	UaF/DF	[15]	356	0.24 / 0.68	0.49 / 1.37	✗	✗	✗	✗	✓
17	UaF/DF	[15]	351	0.23 / 0.64	0.49 / 1.41	✗	✗	✗	✗	✓
18	format string attack	[15]	92	0.23 / 2.47	0.47 / 5.06	✗	✗	✗	✗	✓
19	buffer overflow	[15]	287	0.23 / 0.82	0.54 / 1.88	✗	✗	✗	✗	✓
20	format string attack	[15]	168	0.23 / 2.47	0.47 / 5.06	✗	✗	✗	✗	✓

TABLE II: Experimental results on real-world programs.

Dataset	Total	Filtered	Valid	Success	Miss	TP
CWE	117	48	69	66	3	95.65%

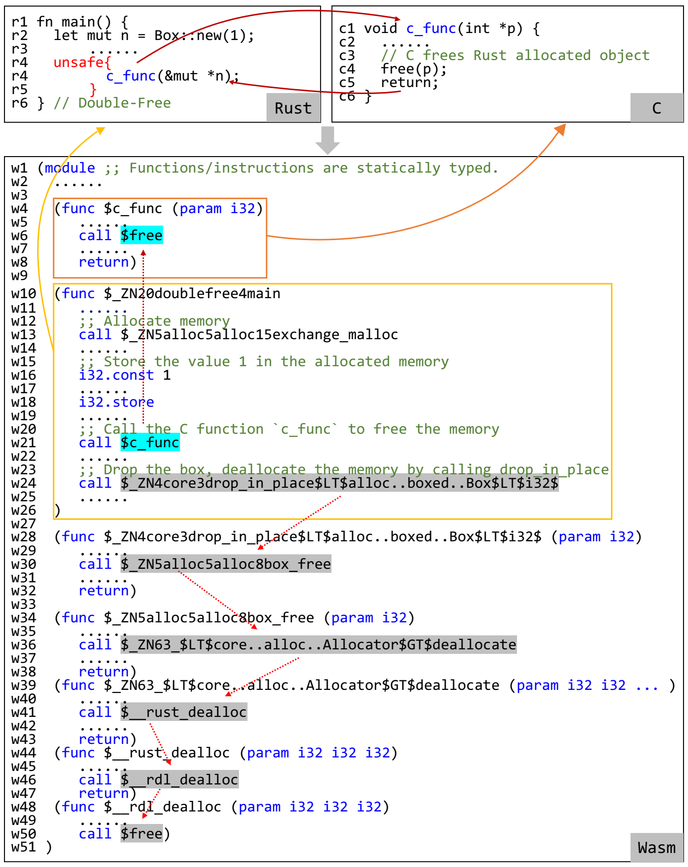
C14 standard (3 cases); or 2) they lack necessary information such as incomplete data structures and undefined functions (45 cases). Among the remaining 69 valid test cases, 66 were successfully detected by ACORN whereas 3 were not. Hence, the true positive (TP) is 95.65% ($\frac{Success}{Valid}$). To further investigate the root causes for the 3 failed cases, we performed a manual analysis for them. This analysis revealed two important reasons: 1) C/C++ and Rust features unsupported by Wasm; or 2) limitations of existing program analysis.

7.5. Performance of ACORN

TABLE I (the 5th and 6th columns) presents the performance of ACORN, including: 1) time for converting the source code to ACRONIR (Conversion Time); and 2) time for program analysis on ACRONIR (Analysis Time). We ran each test case 100 rounds, then calculated the average. Experimental results demonstrated that ACORN is efficient in detecting vulnerabilities in multilingual Rust applications: the time spent on code conversion into ACORN is around 0.2 seconds for each case, with 1.2 milliseconds per line of code; whereas the analysis time is about 0.5 seconds for each case, with 2.5 milliseconds per line of code. Moreover, the conversion time is less than the analysis time, thus the overhead of the conversion is negligible.

7.6. Comparison with Peer Tools

For peer comparison, we used the closest, state-of-the-art baselines we can find: CRust [15], a cross-language program analysis framework for Rust and C by converting C to MIR; and FFIChecker [16], designed to detecting memory management issues caused by interaction of Rust and C/C++ via FFIs. We did try to compare with several other tools that claimed to support cross-language analysis. Unfortunately, few of them are publicly available online (e.g., Truffle [29]) and actually usable.



```
r1 fn main() {
r2   let mut n = Box::new(1);
r3      ......
r4   unsafe{
r4          c_func(&mut *n);
r5       }
r6 } // Double-Free
```

```
c1 void c_func(int *p) {
c2    ......
c3    // C frees Rust allocated object
c4    free(p);
c5    return;
c6 }
```

```
w1 (module ;; Functions/instructions are statically typed.
w2   ......
w3
w4   (func $c_func (param i32)
w5      ......
w6      call $free
w7      ......
w8      return)
w9
w10  (func $_ZN20doublefree4main
w11     ......
w12     ;; Allocate memory
w13     call $_ZN5alloc5alloc15exchange_malloc
w14     ......
w15     ;; Store the value 1 in the allocated memory
w16     i32.const 1
w17     ......
w18     i32.store
w19     ......
w20     ;; Call the C function `c_func` to free the memory
w21     call $c_func
w22     ......
w23     ;; Drop the box, deallocate the memory by calling drop_in_place
w24     call $_ZN4core3drop_in_place$LT$alloc..boxed..Box$LT$i32$
w25     ......
w26  )
w27
w28  (func $_ZN4core3drop_in_place$LT$alloc..boxed..Box$LT$i32$ (param i32)
w29     ......
w30     call $_ZN5alloc5alloc8box_free
w31     ......
w32     return)
w33
w34  (func $_ZN5alloc5alloc8box_free (param i32)
w35     ......
w36     call $_ZN63_$LT$core..alloc..Allocator$GT$deallocate
w37     ......
w38     return)
w39  (func $_ZN63_$LT$core..alloc..Allocator$GT$deallocate (param i32 i32 ... )
w40     ......
w41     call $__rust_dealloc
w42     ......
w43     return)
w44  (func $__rust_dealloc (param i32 i32 i32)
w45     ......
w46     call $__rdl_dealloc
w47     return)
w48  (func $__rdl_dealloc (param i32 i32 i32)
w49     ......
w50     call $free)
w51 )
```

Figure 7: Double-Free bugs in a Rust-C program and the translated AcronIR code.

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 ACRONIR code snippet in Figure 7 for comparison.

As shown in Figure 7, Rust calls a foreign function, `c_func` defined in the external C program, and this behavior is marked by the `unsafe` keyword because the Rust compiler can not cross the boundary imposed by the FFI to analysis C code. Thus, 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. Such vulnerabilities caused by cross-language interactions, cannot be detected by single-language program analysis tools, because the isolation brought about by language differences (including language features and memory management strategies) allows them to speak only of external code written in other languages as black boxes.

ACORN eliminates the isolation due to language differences by proposing a unified specification language, ACRONIR and translating both Rust and C code to it. As we can see in Figure 7, the Rust-C program has been translated into a ACRONIR `module`, in which, the C function `c_func` (line w4-w8), the Rust function `fn main` (line w10-w26) and other library functions called are all defiend. Furthermore, 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 remove the boundary imposed by FFI. As a result, performing program analysis on ACORN makes it easy to obtain a complete call graphs with a unified trace of memory allocation and release track memory allocation and release uniformly, as shown in Figure 7, the memory occupied by the Box object was freed twice (line w21 and line w24).

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 the first step towards defining a unified and effective static analysis framework for multilingual Rust applications.

More comprehensive types of vulnerabilities. While ACORN is effective in detecting memory-related vulnerabilities in multilingual Rust applications, its capabilities can be extended to cover other types of vulnerabilities as well. In particular, enhancing ACORN with the ability to check for concurrency vulnerabilities will make it more comprehensive. To achieve this, we can leverage the latest research advancements, such as deadlock detection [**?**], data race detection [**?**], and type confusion detection [**?**], which are relevant to concurrency security. It is important to note that this remains an area for future work.

More Source Language. Although we focus on Rust combined with C, the idea of ACORN and the ACRONIR model can be extended to other cross-language scenarios. Especially, the converter and the static analyzer in ACORN are designed to be independent of each other. Therefore by changing the converter, our approach can be adapted to analyze other FFIs, as long as they support the compilation to Wasm, e.g., languages such as Python, Javascript, and Go.

9. RELATED WORK

In recent years, there has been a significant amount of research on both Rust security, Wasm security, and multilingual program security. However, this work stands for a novel contribution to these fields.

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 focus on security vulnerabilities and `unsafe` Rust. Qin et al. [4] conducted an empirical study of memory and concurrency security vulnerabilities in Rust applications. Xu et al. [73] conducted an in-depth study of 186 memory security-related CVEs and proposed a taxonomy. Astrauskas et al. [74] studied the use of `unsafe` in 31867 Rust crates and summarized the usage scenarios of `unsafe`. SafeDrop [11], Mirchecker [14], Rupair [12] and Rudra [13] all perform vulnerability detection based on program analysis. However, the above work is limited to pure Rust code, which cannot analyze other languages in multilingual applications, and therefore cannot detect vulnerabilities caused by the interactions of Rust with other languages.

Program Analysis for Wasm. Program analysis for Wasm has been extensively researched and several notable studies have contributed to this field. Haas et al. [54] proposed an operational semantics and type system for ensuring the safety of Wasm programs. Stiévenart et al. [75] developed an information flow analysis algorithm, while Lopes et al. [76]] introduced a vulnerability detection framework using a code property graph. Chen et al. [58] proposed a fuzzing framework specifically for Wasm smart contracts. Additionally, Szanto et al. [29] and Fu et al. [30] conducted taint analysis to track data propagation. However, these studies do not focus on detecting bugs originated from interaction between Rust and external code written in other languages in multilingual Rust programs. In contrast, we proposes the unified specification language, ACRONIR based on Wasm, to perform cross-language program analysis leveraging Wasm program analysis algorithms and tools.

Multilingual Applications Security. There have been many studies on the security of multilingual applications. Mergendahl et al. [9] introduce the threat model for analyzing cross-language attacks on Rust and Go. Li et al. [16] design and implement a pass in LLVM to detect cross-language memory vulnerability in Rust/C. Jiang et al. [77] present a dynamic analysis framework, PolyCruise, to perform information flow analysis on multilingual applications. Costanzo et al. [78] formally verify end-to-end security of software systems that consist of both C and assembly. Hu et al. [15] 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. Our work differs from the above efforts in that we propose a completely unified specification language, ACRONIR based on Wasm, which has a promising ability to perform valid program analysis across the boundaries between Rust and external code written in other languages.

10. CONCLUSION

This paper presents ACORN, a novel unified program analysis framework for analyzing multilingual Rust programs. At the core of ACORN is a novel unified specification language ACRONIR based on Wasm, as the compilation from Rust multilingual programs to it on which cross-language program analysis can be performed. Moreover, ACORN take advantages of existing translation tools and Wasm program analysis algorithms. We have implemented a prototype system for ACORN and conducted extensive experiments. Experimental results show that ACORN can effectively detect common security vulnerabilities across Rust and C. This work represents a new step towards securing multilingual Rust applications, making the promise of a safe system language a reality.

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] 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.

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

[6] 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.

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

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

[9] S. Mergendahl, N. Burow, and H. Okhravi, "Cross-language attacks," in *Proceedings 2022 Network and Distributed System Security Symposium*. San Diego, CA, USA: Internet Society, 2022.

[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*, Jun. 2020, pp. 246–257.

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

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

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

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

[15] 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.

[16] Z. Li, J. Wang, M. Sun, and J. C. S. Lui, "Detecting cross-language memory management issues in rust," in *Computer Security – ESORICS 2022*, V. Atluri,

R. Di Pietro, C. D. Jensen, and W. Meng, Eds. Cham: Springer Nature Switzerland, 2022, vol. 13556, pp. 680–700.

[17] "Llvm language reference manual," https://llvm.org/docs/LangRef.html.

[18] "Mir (mid-level ir)," https://rustc-dev-guide.rust-lang.org/mir/index.html.

[19] "Compiling from rust to webassembly," https://developer.mozilla.org/en-US/docs/WebAssembly/Rust_to_Wasm.

[20] "Compiling a new c/c++ module to webassembly," https://developer.mozilla.org/en-US/docs/WebAssembly/C_to_Wasm.

[21] "Webassembly · golang/go wiki," https://github.com/golang/go/wiki/WebAssembly.

[22] "Compile python to webassembly," https://pythondev.readthedocs.io/wasm.html.

[23] "Bytecodealliance/javy: Js to webassembly toolchain," https://github.com/bytecodealliance/javy.

[24] "Webassembly high-level goals - webassembly," https://webassembly.org/docs/high-level-goals/.

[25] "Security - webassembly," https://webassembly.org/docs/security/.

[26] 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*. Barcelona Spain: ACM, Jun. 2017, pp. 185–200.

[27] 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.

[28] 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.

[29] A. Szanto, T. Tamm, and A. Pagnoni, "Taint tracking for webassembly," Jul. 2018.

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

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

[32] "Types," https://doc.rust-lang.org/reference/types.html.

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

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

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

[36] 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.

[37] 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.

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

[39] "Redox-os / tfs · gitlab," https://gitlab.redox-os.org/redox-os/tfs.

[40] "Smoltcp," https://github.com/smoltcp-rs/smoltcp, Jul. 2023.

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

[42] "Tikv/tikv," TiKV Project, Jul. 2023.

[43] "Blockchain infrastructure for the decentralised web — parity technologies," https://www.parity.io/.

[44] "Oxide - home," https://oxidemod.org/.

[45] "Official site — second life - virtual worlds, virtual reality, vr, avatars, and free 3d chat," https://secondlife.com/.

[46] "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/.

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

[48] "Rust for linux," https://github.com/Rust-for-Linux.

[49] "Going public launch bug · issue #150 · webassembly/design."

[50] "Roadmap - webassembly," https://webassembly.org/roadmap/.

[51] "Webassembly core specification," https://www.w3.org/TR/wasm-core-1/.

[52] "World wide web consortium (w3c) brings a new language to the web as webassembly becomes a w3c recommendation," https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en.

[53] "Standardizing wasi: A system interface to run webassembly outside the web – mozilla hacks - the web developer blog," https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface.

[54] 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*. Barcelona Spain: ACM, Jun. 2017, pp. 185–200.

[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)*. Barcelona, Spain: IEEE, Jul. 2022, pp. 543–553.

[56] "Homepage — wasmcloud."

[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*. Virtual South Korea: 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*. San Diego, CA: Internet Society, 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*. London United Kingdom: ACM, 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] “Torch2424/wasmboy: Game boy / game boy color emulator library, written for webassembly using assemblyscript. demos built with preact and svelte.” 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, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems*, vol. 32. Curran Associates, Inc., 2019.

[67] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, “Array programming with numpy,” *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020.

[68] “Java native interface specification contents,” https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html.

[69] “Ffi - the rustonomicon.”

[70] “Python/c api reference manual,” https://docs.python.org/3/c-api/index.html.

[71] “Crates.io: Rust package registry,” https://crates.io/.

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

[73] 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*. London UK: ACM, Jun. 2020, pp. 763–779.

[74] 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.

[75] 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.

[76] 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.

[77] W. Li, M. Jiang, X. Luo, and H. Cai, “Polycruise: A cross-language dynamic information flow analysis.”

[78] D. Costanzo, Z. Shao, and R. Gu, “End-to-end verification of information-flow security for c and assembly programs,” in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Santa Barbara CA USA: ACM, Jun. 2016, pp. 648–664.