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

Shuang Hu, Baojian Hua*, Lei Xia, and Yang Wang*
School of Software Engineering, University of Science and Technology of China, China
{guangan, xialeics}@mail.ustc.edu.cn {bjhua, angyan}@ustc.edu.cn*
* Corresponding authors.

***Abstract*—Rust is a new safe system programming language enforcing safety guarantees by novel language features, a rich type system, and strict compile-time checking rules, and thus has been used extensively to build system software. For 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, existing security studies on Rust only focus on pure Rust code but cannot analyze either the native C code or the Rust/C interactions 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 CRust, a unified program analysis framework across Rust and C, which enables program analyses to understand the semantics of C code by translating Rust and C into a unified specification language. The CRust framework consists of three key components: (1) a unified specification language CRustIR, which is a strong-typed low-level intermediate language suitable for program analysis; (2) a transformation to build models of C code by converting C code into CRustIR; and (3) program analysis algorithms on CRustIR to detect security vulnerabilities. We have implemented a software prototype for CRust, and have conducted extensive experiments to evaluate its effectiveness and performance. Experimental results demonstrated that CRust can effectively detect common memory security vulnerabilities caused by the interaction of Rust and C that are missed by state-of-the-art tools. In addition, CRust is efficient in bringing negligible overhead (0.23 seconds on average).

Keywords*—*Rust, Security, Multilingual Program Analysis

I. Introduction

Software failures or vulnerabilities may lead to devastating consequences or losses, especially in security-critical scenarios [1]. Historically, C/C++ has been the dominant language for building software infrastructures such as operating kernels and network protocol stacks. Although C/C++ is both flexible and efficient, it also offers "flexibility" in introducing subtle vulnerabilities, due to the lack of necessary security checks. As a result, despite decades of research efforts, memory security vulnerabilities are still prevalent in real-world system software developed with C/C++ [2]. For example, according to a report by Microsoft Security Response Center (MSRC), 70% of security patches from 2006 to 2018 addressed memory security vulnerabilities [3]. As another example, approximately 70% of the high severity security vulnerabilities in the Chromium project are memory security vulnerabilities [4].

Rust [5] is a new generation of safe system programming language, designed to address the security issues of traditional languages such as C/C++. Rust drew on practical experience and research findings in the field of programming languages over the past several decades and thus introduced a set of novel security-related language features, such as the ownership model [6], borrow [7], reference [8], and automatic lifetime-based memory management [9]. Rust also provided strict security checking rules, which are enforced either statically or dynamically. These language features, as well as checking rules, guarantee that Rust programs are free of memory security vulnerabilities, such as use-after-free and double-free [10], without sacrificing runtime efficiency.

While Rust provided strong security guarantees, the overly strict restrictions also make it difficult or even impossible to develop low-level code. To support arbitrary low-level programming and to provide more flexibility to developers, Rust introduced the `unsafe` sub-language [11] , which is essentially a security loophole by allowing unsafe operations in code blocks marked with the `unsafe` keyword. Existing studies, unfortunately, have demonstrated that improper use of the `unsafe` sub-language can lead to security vulnerabilities [10] [12] [13]. As a result, although Rust represents a significant step forward in the field of safe system programming, security vulnerabilities in Rust pose a grand challenge to Rust security, and thus may defeat Rust as a safe language. To address the Rust security challenge, many research efforts have been conducted (e.g., security empirical study [10] [12] [13], vulnerability prevention [14] [15] [16] [17], vulnerability detection [18] [19] [20] [21], and formal verification [22] [23] [24] [25]), in which program analysis is extensively used [15] [18] [19] [20] [21] [26] [27]. As a result, the effectiveness of prior Rust security studies largely depends on the precision of program analysis. High-precision program analysis not only reduces the false positives and false negatives of vulnerability detection, but also improves the effectiveness of vulnerability prevention. For example, Rudra [20], one state-of-the-art vulnerability detection framework detecting 264 previously unknown memory security vulnerabilities, employed an inter-procedural program analysis algorithm.

Unfortunately, the horizon of existing Rust program analysis frameworks is limited to code written in Rust only, and yet many real-world Rust applications are multilingual, containing a mixture of Rust and C code. For example, in the source code of Firefox, a flagship application of Rust from Mozilla, the percentage of Rust code is 9.6%, while the percentage of C code is 13.9% [28]. Typically, existing Rust program analyses treat C code in Rust applications as black boxes, and make either optimistic or pessimistic assumptions about the C code. On one hand, the optimistic assumption treats the C code as a nop, and thus ignores potential vulnerabilities in the C code. On the other hand, the pessimistic assumption assumes anything can happen in the C code, and thus always issues errors which may lead to high false positives. For example, MirChecker [27], an abstract interpretation framework on Rust, will always issue errors when encountering C code. As a result, existing Rust program analysis cannot detect errors caused by the C code or interactions of Rust and C, which hinders the broad applications of such analysis.

In this paper, we propose CRUST, a unified program analysis framework for Rust across the boundary of Rust and C. This framework enables existing Rust program analyses to model and understand the semantics of both Rust and C code in multilingual Rust applications, by simultaneously translating both Rust and C into a unified specification language. The CRUST framework consists of three main components: (1) a unified and formal specification language CRUSTIR, which is a strongly typed low-level intermediate representation (IR); (2) a transformation from C to CRUSTIR, which builds models of C code concisely and precisely; and (3) program analysis algorithms on CRUSTIR to detect security vulnerabilities. In order to define CRUST rigorously, we also formally define the syntax and operational semantics of CRUSTIR, as well as the conversion rules for translating C code into CRUSTIR.

Such a design brings three significant advantages to our unified program analysis framework: (1) expressiveness; (2) simplicity; and (3) usefulness. First, CRUST is expressive because CRUSTIR is a strongly typed low-level intermediate representation that accurately models the semantics of Rust and C code, which may improve the precision of the analysis greatly. Second, the design of CRUST simplifies the implementation of CRUST, as both Rust and C code can be translated to CRUSTIR more straightforwardly. Finally, the utility of CRUST is good because not only new program analysis algorithms can be developed, but also most existing Rust program analysis algorithms can be ported to CRUST with minimal or even no modifications.

We have conducted experiments to evaluate the effectiveness and performance of CRUST. First, to evaluate the effectiveness of CRUST, we testify CRUST on two datasets: 1) a micro-benchmark containing common memory security vulnerabilities caused by the interactions of Rust and C; and 2) real-world vulnerability sets. Experimental results demonstrated that CRUST can effectively detect these vulnerabilities, whereas existing analysis tools detected none of them. Furthermore, CRUST is efficient by incurring a runtime overhead of 0.23 seconds on average.

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

- **A unified program analysis framework that works across Rust and C.** We present CRUST, the first program analysis framework working across the boundary between Rust and C, by formal definitions of the two components of CRUST: a unified specification language CRUSTIR, and conversion rules from C to CRUSTIR.
- **Prototype implementation of CRUST.** We implemented a prototype of CRUST, by translating Rust and C code into CRUSTIR, and by porting existing Rust analysis to CRUSTIR for multilingual program analysis.
- **Evaluation of CRUST.** We conducted extensive experiments to evaluate the effectiveness and performance of CRUST. Experimental results demonstrated that CRUST can effectively detect vulnerabilities across Rust and C with negligible additional overhead.

Outline. The rest of this paper is organized as follows. Section II introduces the background knowledge and motivations for this work. Section III presents a formal definition of the syntax and operational semantics of CRUSTIR. Section IV formally describes the translation rules from C to CRUSTIR. Section V presents a prototype implementation. Section VI presents the evaluations we conducted. Section VII discusses limitations and future work. Section VIII describes related work, and Section IX concludes.

II. BACKGROUND AND CHALLENGES

In this section, we first present the necessary background knowledge (Section II-A) and challenges (Section II-B) for this work, then give a threat model (Section II-C).

A. The Rust Programming Language

To achieve the design goals of security and efficiency, Rust introduced a number of security-oriented language features, a strong type system, and an ownership model. First, Rust is a multi-paradigm programming language providing a rich set of safe functional programming mechanisms [29], such as pattern matching and variable immutability, which prevent memory security vulnerabilities. Second, Rust incorporates a strong type system [30] and strict type checking [31] to ensure type safety. In addition, Rust also supports polymorphic types and local type derivation, to alleviate type constraints and improve code reusability. Finally, Rust introduces an ownership model [6]. On one hand, by statically enforcing ownership rules at compile time, Rust not only eliminates memory security vulnerabilities commonly found in C/C++, but also avoids the problem of race conditions in concurrency. On the other hand, Rust adopts an ownership-based automatic memory management mechanism, which avoids the overhead of garbage collection and ensures high runtime efficiency.

Due to its successful combination of security and efficiency, Rust has gained more popularity in recent years. According to a developer survey conducted by Stack Overflow [32], Rust has been rated as the “most popular programming language” for five consecutive years, with 86.98% of developers are using, or considering using, Rust. In the meanwhile, Rust has been used successfully to build software infrastructures, such as operating system kernels [33] [34] [35] [36] [37], Web browsers [38], file systems [39] [40], cloud services [41], network protocol stacks [42], language runtime [43], databases [44], and blockchains [45]. Rust is also gaining more adoptions by the industry. Among others, Microsoft [46], Google [47], and even Linux [48] are beginning to use Rust for the development of system software.

To offer more programming flexibility, Rust provided an `unsafe` sub-language allowing `unsafe` operations, which is essentially a security loophole by bypassing the compiler's security checks. Hence, improper uses of `unsafe` may lead to security vulnerabilities, forming new attack surfaces [49]. For example, `unsafe` Rust allows arbitrary pointer arithmetics, which may result in arbitrary memory address reading/writing, further leading to vulnerabilities such as segmentation faults or buffer overflows [50]. In fact, according to prior studies [10], all memory security vulnerabilities are related to `unsafe` Rust .

B. Challenges for Security Study of Multilingual Rust

The foreign function interface (FFI) [51] is an important component of Rust, which enables Rust to interact with external languages. Figure 1 shows a simple yet illustrative FFI sample usage in Rust. First, FFI function declarations are marked with the `extern` keyword (line 1 of Figure 1(a)), thus the Rust compiler can compile the Rust code based solely on such external declarations (line 8 of Figure 1(a)). To make the interaction with C code easy, the Rust compiler follows the C-ABI, thus external C functions (e.g., line 2 of Figure 1(b)) can be compiled separately then linked (either statically or dynamically) together with the Rust code. Hence, Rust FFI fully leveraged the benefit of the separation compilation model of C, simplifying legacy code reuse considerably.

Although FFI brings great programming capabilities and huge development potential to Rust, FFI is inherently `unsafe` in Rust (note the `unsafe` keyword at line 8 of Figure 1(a)), as the Rust compiler does not perform any security checking on the external C functions being called. Such a lack of security checking, unfortunately, can easily lead to security vulnerabilities. Figure 1 demonstrates a Use-after-Free (UaF) vulnerability due to such interactions. First, Rust allocates a heap object (line 6 of Figure 1(a)), then passes this pointer to the external C function (line 8). Then, the external C function frees the object (line 5 of Figure 1(b)). Returning from the C function, the Rust code (line 10 of Figure 1(a)), when accessing the pointer, triggers a Use-after-Free (UaF) vulnerability, which is difficult to diagnose as the root cause is located in the C code instead of Rust. Although we have made use of UaF for illustration, it is not a unique vulnerability. Indeed, as Rust and C code share the same process address space without any isolation, any memory vulnerabilities, such as UaF, double frees, and nullable dereferencing, can occur in such multilingual applications.

It's challenging to detect security vulnerabilities across Rust and C in multilingual applications, for two key reasons. First, the Rust compiler does not perform any checking on external C functions, due to the dramatic discrepancies between Rust and C. Instead, the Rust compiler always make pessimistic decisions by marking external functions unsafe. Second, existing Rust analysis studies and tools can only handle pure Rust code and treat external C functions as black boxes [18]

```
1  extern "C" {
2    fn C_fn(obj_ptr: i64);
3  }
4  fn Rust_fn(){
5    // "heap_obj" points to a cell allocated by Rust
6    let mut heap_obj = vec![1,2,3];
7    // Pass "heap_obj" to a C function
8    unsafe{C_fn(heap_obj);}
9    // Use-after-free vulnerability is triggered
10   heap_obj[0] += 5;
11 }
```

(a) Rust function that invokes an external C function, passing a Rust pointer wrongfully freed by C.

```
1 // Frees object it dose not own
2 void C_fn(int64_t obj_ptr){
3   int64_t *addr = (int64_t *)obj_ptr;
4   //C frees Rust allocated object!
5   free(addr);
6 }
```

(b) C function that accepts and free a Rust pointer, leading to a Use-after-Free (UaF) vulnerability in Rust.

Figure 1. Sample code illustrating a Use-after-Free vulnerability across Rust and C.

[19] [20]. At the same time, external function calls are one of the most common `unsafe` operations in Rust, accounting for 22.5% of all `unsafe` function calls [13]. As a result, a large number of external function calls to C in Rust programs cannot be statically analyzed by either the Rust compiler or Rust analyses, which poses a threat to the security and reliability of the software infrastructures in Rust. To this end, it is a grand challenge to address the security issues in multilingual Rust applications.

C. Threat Model

This work focuses on the study of a unified program analysis for multilingual applications across Rust and C. Therefore, we make the following assumptions in the threat model.

We assume that the host environment for Rust application executions is safe, including the underlying hardware, operating system, compiler, and linker. A large number of protection mechanisms have been proposed and standardized, including Software Guard eXtensions (SGX) [52], Address Space Layout Randomization (ASLR) [53], among others. In addition, many security studies have been conducted in this direction [54]. It should be noted that operating systems and compilers security studies are independent of and thus orthogonal to the study in this work, and these research fields can also benefit from the research progress in this work.

We assume that pure Rust code, including all `unsafe` operations except for external C function calls, is safe and will not pose a security threat to the application. As there are already a significant amount of studies [27] [19] [20] [55] [56] [57] on either pure Rust or unsafe Rust without external function calls, such an assumption is reasonable in reality.

We assume that the external C functions Rust interacts with are unreliable and thus vulnerable. For example, if the external C functions being called, through Rust FFI, are vulnerable, attackers can control the C code so as to perform further arbitrary attacks, eventually corrupting the whole application.

III. A Unified Specification Language

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

This section will answer the first question, by presenting the specification language design rationale (Section III-A), the syntax (Section III-B), as well as the operational semantics (Section III-D) for this specification language. Then, in Section (Section IV), we will answer the second question by presenting the conversion rules from C to CRustIR.

A. Specification Language Design Rationale

To design the specification language, three design options are feasible: (1) C-oriented; (2) Rust-oriented; or (3) completely neutral.

First, the original C code, in some sense, is the most precise specification as it does not lose any source information. However, the original C code is not an ideal specification for two reasons: first, if the original C code is used as the specification, it requires significant effort to modify all existing Rust analyses so that they can understand the semantics of C code, which is not portable and requires a complete rewrite of the existing analysis. Second, C code does not have the security features of Rust, such as ownership and lifetime. Therefore, Rust analyses cannot perform the important security checks, such as borrow checking [58], on C. Hence, Rust analyses cannot provide the security guarantees to C code as they provide for Rust code.

The second option is to use a Rust-oriented specification language. To be specific, Rust has two key intermediate representations in its compiler: 1) abstract syntax trees (AST); and 2) middle intermediate representations (MIR) [59]. AST is a tree-structured representation of the source program. While AST keeps the source program information precisely, it lacks data flow information, control flow information, and type information after elaboration, which are indispensable for precise program analyses, such as flow-sensitive analysis or inter-procedural analysis. Hence, AST is not suitable as a specification language.

The third option is to use a source-neutral low-level language as the specification language. For example, we can make use of x86, ARM, or WebAssembly [60], the newest general-purpose binary format initially designed for Web, to serve as the general specification language. To this end, we should translate both Rust and C in the multilingual application to this neutral language, then conduct program analysis or verification on it. Although this approach is general to support any source language, it has two key limitations. First, the translation incurs considerable efforts, as we have to convert two languages, i.e. C and Rust, instead of one, to the neutral low-level language. Second, translation from Rust or C to the neutral low-level language may lose important information, which may further make the detection of vulnerabilities difficult. For example, x86 or ARM is untyped whereas WebAssembly only contains simple scalar types. Hence, it is challenging to recover meaningful type errors to end-users, from these low-level languages.

To this end, the approach that we propose is to define the specification language, dubbed as CRustIR, based on MIR. MIR is a control-flow graph-based intermediate representation, and thus has three distinct properties which make it suitable for program analysis. First, MIR offers a simplified core representation of Rust by removing Rust's superficial syntactic forms. For example, all control constructs in Rust such as `if` or `for` are removed by lowering them down to explicit control transfers. Such removals not only simplifiy program analysis on MIR, but also makes MIR an ideal compilation target for other languages like C. Second, MIR is strongly typed in that all variables in MIR are explicitly type-tagged. Strong typed intermediate representations make many static program analyses, such as pointer analysis, much more precise [61]. Finally, MIR explicitly marks variable lifetimes with `storageLive` and `storageDead` statements. This property makes borrow checking, one of Rust's most important static checks, feasible on MIR.

To summarize, our design rationale of the CRustIR specification language based on MIR has three key advantages: 1) expressiveness; 2) generality; and 3) flexibility. First, this specification language is expressive. The primitives in CRustIR are similar to those contained in both intermediate representations of C and Rust, so that we can faithfully compile C code to this specification language, thus precisely model the behavior of C code.

Second, existing Rust program analysis on the MIR can be easily carried over to our specification language CRustIR, as these analyses can distill all the needed information from this specification language. Furthermore, more precise and comprehensive Rust program analysis algorithms can also be constructed for this specification language.

Finally, CRustIR is flexible to support manual construction of models for C code. Programmers can easily specify a model of C code by writing a piece of Rust code which is then compiled into CRustIR.

B. Syntax

In this section, we present the syntax for the CRustIR specification language, as summarized in Figure 2. Each program P consists of a list of functions F, where the notation $\overrightarrow{F}$ stands for zero or more of function F, we sometimes also write $F_1, \ldots, F_n$, for $n \geq 0$.

Each function F consists of a list of arguments $\overrightarrow{o : \tau}$, a return type τ, a list of local variable declarations $\overrightarrow{D}$, followed by a list of basic blocks $\overrightarrow{b : B}$. These syntactic entities have some

$$
\begin{array}{llll}
\text{Constant} & c & & \\
\text{Bid} & b & \in & \mathbb{N} \\
\text{Variable} & x & \in & \{x_0, x_1, x_2, \ldots\} \\
\text{Fn} & f & \in & \{f_0, f_1, f_2, \ldots\} \\
\text{Type} & \tau & ::= & \texttt{bool} \mid \texttt{int} \mid \texttt{unit} \\
 & & \mid & \texttt{Vec}\langle\tau\rangle \mid \texttt{Tuple}(\vec{\tau}) \mid \texttt{Array}[\tau, n] \\
\text{Value} & v & ::= & \texttt{true} \mid \texttt{false} \mid n \mid () \\
 & & \mid & (\vec{v}) \mid [\vec{v}] \mid \texttt{vec}[\vec{v}] \mid l \\
\text{BinOp} & \oplus & ::= & + \mid - \mid \times \mid / \mid < \mid == \mid \ldots \\
\text{UniOp} & \ominus & ::= & ! \mid - \\
\text{Operand} & o & ::= & \texttt{const}\ c \mid \texttt{move}\ p \mid \texttt{copy}\ p \\
\text{Place} & p & ::= & x \mid *x \mid p.n \mid p[x] \\
\text{Rvalue} & r & ::= & o \mid \&[\texttt{mut}]p \mid \ominus\, o \\
 & & \mid & x_1 \oplus x_2 \mid o\ \texttt{as}\ \tau \\
\text{Statement} & s & ::= & x = r \mid \texttt{storageLive}(x) \\
 & & \mid & \texttt{storageDead}(x) \\
\text{Terminator} & t & ::= & \texttt{goto} \rightarrow b \\
 & & \mid & \texttt{switch}(x) \rightarrow (\overrightarrow{n : b})\ \texttt{default} \rightarrow b \\
 & & \mid & \texttt{assert}(o) \rightarrow b \mid \texttt{drop}(x) \rightarrow b \\
 & & \mid & p = f(\vec{o}) \rightarrow b \mid f(\vec{o}) \mid \texttt{return}\ x \\
\text{Block} & B & ::= & \vec{s}\ t \\
\text{Declaration} & D & ::= & \texttt{let}\ [\texttt{mut}]\ x : \tau = r \\
\text{Function} & F & ::= & \texttt{fn}\ f(\overrightarrow{o : \tau}) \rightarrow \tau\ \{\vec{D};\ \overrightarrow{b : B}\} \\
\text{Program} & P & ::= & \vec{F}
\end{array}
$$

Figure 2. Syntax of CRUSTIR

characteristics specific to Rust. For example, each variable declaration D can be marked with an optional [mut] keyword, indicating the variable x is mutable, as all variables in Rust are immutable by default.

A basic block $b : B$ has a unique identifier b, followed by a body B, which contains a list of statements s and a single terminator t. A statement s may be an assignment $x = r$, or Rust-specific `storageLive` and `storageDead` marking the start and end of a variable's lifetime, respectively. Like the control-flow graph in compilers, controls in block body B can only exit from the last entity, a terminator t, which has several distinct syntactic forms: 1) an unconditional jump `goto`, here the notation $\rightarrow b$ stands for the target for this jump is another basic block b; 2) a switch jump `switch` (with a `default`); 3) an assertion `assert`; 4) a deletion `drop`; and 5) function invocations and returns.

A place p can appear at the left side of an assignment, whereas a right value r can only appear at the right side. A place p stands for a location which can be assigned to, which includes a variable x, pointer references, tuple field selection, or array elements. A right value r consists of operands o, reference &, binary or unary operations, and type castings. Both right values r and operands o have Rust-specific syntactic features. For example, a place p in an operand o can be marked by either `move` or `copy`, which represents the move or copy semantics in Rust [62] [63], respectively. In the meanwhile, the mutable reference of a place p can be obtained by a place-of & operation.

A type τ consists of representative Rust types, including atomic types such as `bool`, and aggregate types such as vectors, tuples, and arrays.

Finally, a value v is a syntactic entity that a right value r may evaluate to, which consists of not only atomic values such as boolean, integer, or unit constants, but also aggregate values for vectors, tuples, or arrays. Furthermore, l is a special value indicating the memory address of a heap value, which will be discussed further in the following.

TABLE I
NOTATIONS FOR OPERATIONAL SEMANTICS OF CRUSTIR

Notations	Descriptions
Γ	A global environment, representing a program.
Θ	A function call stack.
$C = B, \Theta, \Psi$	A CRUST machine configuration: B: the statements or terminator to be executed within the current basic block. Θ: the function calls stack of the current program. Ψ: the global heap mapping addresses to values.
$\langle b_r, x_r, \Sigma, F\rangle$	The stack frame in the function call stack: b_r: the id of basic block to be executed after the current function returns. x_r: the place where the returned value of the current function must be written. Σ: the variable store of the current function. F: the body of current function.

C. Memory Model

Before defining the operational semantics, we first define a memory model for CRUSTIR to describe the state of program. To be specific, the memory model $\mathcal{M} = (\Sigma, \Psi)$ we will be using consists of two components: the first component $\Sigma : x \mapsto l$ is a store, mapping a program variable x to its heap address l.

The second component $\Psi : l \mapsto v$ is a heap, mapping a memory address l to its containing value v. To this end, we have stored all values v in heap Ψ, and thus a variable x always contain the heap address l of its corresponding value v. Such a memory model design not only models the semantics of address-of operator & faithfully, but also makes the formalization of move/copy semantics easier.

It should be noted that the store Σ is local, with one for each function, whereas Ψ is global for the entire program. This design decision has two advantages: 1) operations such as reference and dereferencing can be described intuitively, and 2) local variables in different functions may have the same name.

D. Operational Semantics

This section presents the operational semantics of CRUSTIR. To describe control transfer and function call/returns, we will use the auxiliary notations in TABLE I.

The operational semantics is defined in a big-step style and is given by four judgments:

Place: $\boxed{\Sigma, \Psi \vdash p \Downarrow v}$

$$[\text{Var}]\frac{\Sigma(x) = l \qquad \Psi(l) = v}{\Sigma, \Psi \vdash x \Downarrow v} \qquad [\text{Array}]\frac{\Sigma, \Psi \vdash x \Downarrow n \qquad \Sigma, \Psi \vdash p \Downarrow \texttt{Array}[v_0, \dots, v_k] \qquad 0 \leq n \leq k}{\Sigma, \Psi \vdash p[x] \Downarrow v_n}$$

$$[\text{Ptr}]\frac{\Sigma(x) = l}{\Sigma, \Psi \vdash *x \Downarrow l} \qquad [\text{Tuple}]\frac{\Sigma, \Psi \vdash p \Downarrow \texttt{Tuple}(v_0, \dots, v_k) \qquad 0 \leq n \leq k}{\Sigma, \Psi \vdash p.n \Downarrow v_n}$$

Operand: $\boxed{\Sigma, \Psi \vdash o \Downarrow v}$

$$[\text{Const}]\frac{}{\Sigma, \Psi \vdash \texttt{const}\ c \Downarrow c} \qquad [\text{Move}]\frac{\Sigma, \Psi \vdash p \Downarrow v}{\Sigma, \Psi \vdash \texttt{move}\ p \Downarrow v} \qquad [\text{Copy}]\frac{\Sigma, \Psi \vdash p \Downarrow v}{\Sigma, \Psi \vdash \texttt{copy}\ p \Downarrow v}$$

Rvalue: $\boxed{\Sigma, \Psi \vdash r \Downarrow v}$

$$[\text{O}]\frac{}{\Sigma, \Psi \vdash o \Downarrow v} \qquad [\text{Ref}]\frac{\Sigma, \Psi \vdash p \Downarrow v}{\Sigma, \Psi \vdash \&[\texttt{mut}]p \Downarrow v} \qquad [\text{Uop}]\frac{\Sigma, \Psi \vdash o \Downarrow v}{\Sigma, \Psi \vdash \ominus o \Downarrow \ominus v}$$

$$[\text{Bop}]\frac{\Sigma, \Psi \vdash x_1 \Downarrow v_1 \qquad \Sigma, \Psi \vdash x_2 \Downarrow v_2}{\Sigma, \Psi \vdash x_1 \oplus x_2 \Downarrow v_1 \oplus v_2} \qquad [\text{Cast}]\frac{\Sigma, \Psi \vdash o \Downarrow v}{\Sigma, \Psi \vdash o\ \texttt{as}\ \tau \Downarrow v}$$

Statement and terminator: $\boxed{\Gamma \vdash C_0 \Downarrow C_1}$

$$[\text{Declare}]\frac{\Sigma, \Psi \vdash r \Downarrow v \qquad l \text{ is fresh} \qquad \Sigma' = \Sigma[x \mapsto l] \qquad \Psi' = \Psi[l \mapsto v]}{\Gamma \vdash \{\texttt{let}\ [\texttt{mut}]\ x : \tau = r; \overrightarrow{D}\}, \langle b_r, x_r, \Sigma, F\rangle \cdot \Theta, \Psi \Downarrow \{\overrightarrow{D}\}, \langle b_r, x_r, \Sigma', F\rangle \cdot \Theta, \Psi'}$$

$$[\text{Assign}_{\text{move}}]\frac{r = \texttt{move}\ p \qquad \Sigma, \Psi \vdash r \Downarrow v \qquad \Psi' = \Psi[\ \Sigma(x) \mapsto v]}{\Gamma \vdash \{x = r; \overrightarrow{s}; t\}, \langle b_r, x_r, \Sigma, F\rangle \cdot \Theta, \Psi \Rightarrow \{\overrightarrow{s}; t\}, \langle b_r, x_r, \Sigma, F\rangle \cdot \Theta, \Psi'}$$

$$[\text{Assign}_{\text{ref}}]\frac{r = \&[\texttt{mut}]p \qquad \Sigma, \Psi \vdash r \Downarrow v \qquad \Psi' = \Psi[\ \Sigma(x) \mapsto v]}{\Gamma \vdash \{x = r; \overrightarrow{s}; t\}, \langle b_r, x_r, \Sigma, F\rangle \cdot \Theta, \Psi \Downarrow \{\overrightarrow{s}; t\}, \langle b_r, x_r, \Sigma, F\rangle \cdot \Theta, \Psi'}$$

$$[\text{Assign}]\frac{\Sigma, \Psi \vdash r \Downarrow v \qquad \Psi' = \Psi[\ \Sigma(x) \mapsto v]}{\Gamma \vdash \{x = r; \overrightarrow{s}; t\}, \langle b_r, x_r, \Sigma, F\rangle \cdot \Theta, \Psi \Downarrow \{s * t\}, \langle b_r, x_r, \Sigma, F\rangle \cdot \Theta, \Psi'}$$

$$[\text{Assert}]\frac{\Sigma, \Psi \vdash o \Downarrow z}{\Gamma \vdash \{\texttt{assert}(o) \to b;\}, \langle b_r, x_r, \Sigma, F\rangle \cdot \Theta, \Psi \Downarrow F(b), \langle b_r, x_r, \Sigma, F\rangle \cdot \Theta, \Psi}$$

$$[\text{Goto}]\frac{}{\Gamma \vdash \{\texttt{goto} \to b;\}, \langle b_r, x_r, \Sigma, F\rangle \cdot \Theta, \Psi \Downarrow F(b), \langle b_r, x_r, \Sigma, F\rangle \cdot \Theta, \Psi}$$

$$[\text{Drop}]\frac{\Sigma(x) = l \qquad \Psi(l) = v \qquad \Sigma' = \Sigma - [x \mapsto l] \qquad \Psi' = \Psi - [l \mapsto v]}{\Gamma \vdash \{\texttt{drop}(x) \to b;\}, \langle b_r, x_r, \Sigma, F\rangle \cdot \Theta, \Psi \Downarrow F(b), \langle b_r, x_r, \Sigma', F\rangle \cdot \Theta, \Psi'}$$

$$[\text{Switch}_1]\frac{\Sigma, \Psi \vdash o \Downarrow z_i \qquad i \in \{0, \dots, n\}}{\Gamma \vdash \{\texttt{switch}(o) \to [\overrightarrow{z : b}\ \texttt{default} \to b_{n+1}];\}, \langle b_r, x_r, \Sigma, F\rangle \cdot \Theta, \Psi \Downarrow F(b_i), \langle b_r, x_r, \Sigma, F\rangle \cdot \Theta, \Psi}$$

$$[\text{Switch}_2]\frac{\Sigma, \Psi \vdash o \Downarrow v \qquad v \notin \{z_0, \dots, z_n\}}{\Gamma \vdash \{\texttt{switch}(o) \to [\overrightarrow{z : b}\ \texttt{default} \to b_{n+1}];\}, \langle b_r, x_r, \Sigma, F\rangle \cdot \Theta, \Psi \Downarrow F(b_{n+1}), \langle b_r, x_r, \Sigma, F\rangle \cdot \Theta, \Psi}$$

$$[\text{Call}_{\text{Normal}}]\frac{l_j \text{ is fresh} \quad \Sigma, \Psi \vdash o_j \Downarrow v_j \quad \Sigma' = \Sigma[x_j \mapsto l_j, \dots] \quad \Psi' = \Psi[l_j \mapsto v_j, \dots] \quad j \in \{1, \dots, n\} \quad \Gamma(f) = F'}{\Gamma \vdash \{x = f(o_1, \dots, o_n) \to b;\}, \langle b_r, x_r, \Sigma, F\rangle \cdot \Theta, \Psi \Downarrow F'(b_0), \langle b'_r, x'_r, \Sigma', F'\rangle \cdot \langle b_r, x_r, \Sigma, F\rangle \cdot \Theta, \Psi'}$$

$$[\text{Call}_{\text{Panic}}]\frac{l_j \text{ is fresh} \quad \Sigma, \Psi \vdash o_j \Downarrow v_j \quad \Sigma' = \Sigma[x_j \mapsto l_j, \dots] \quad \Psi' = \Psi[l_j \mapsto v_j, \dots] \quad j \in \{1, \dots, n\} \quad \Gamma(f) = F'}{\Gamma \vdash \{f(o_1, \dots, o_n);\}, \langle b_r, x_r, \Sigma, F\rangle \cdot \Theta, \Psi \Downarrow F'(b_0), \langle \_, \_, \Sigma', F'\rangle \cdot \langle b_r, x_r, \Sigma, F\rangle \cdot \Theta, \Psi'}$$

$$[\text{Return}]\frac{\Sigma, \Psi \vdash x \Downarrow v \qquad \Psi' = \Psi[\Sigma'(x_r) \mapsto v]}{\Gamma \vdash \{\texttt{return}\ x;\}, \langle b_r, x_r, \Sigma, F\rangle \cdot \langle b'_r, x'_r, \Sigma', F'\rangle \cdot \Theta, \Psi \Downarrow F'(b_r), \langle b'_r, x'_r, \Sigma', F'\rangle \cdot \Theta, \Psi'}$$

Figure 3. Operational Semantics of CRUSTIR

- $\Sigma, \Psi \vdash p \Downarrow v$: a place p evaluates to a value v;
- $\Sigma, \Psi \vdash o \Downarrow v$: an operand o evaluates to a value v;
- $\Sigma, \Psi \vdash r \Downarrow v$: a right value r evaluates to a value v; and
- $\Gamma \vdash C_0 \Downarrow C_1$: a program state C_0 evaluates to a new state C_1, under the global environment Γ.

The first three judgments are pure and thus have no side effects, whereas the last judgment $\Gamma \vdash C_0 \Downarrow C_1$ may change the state of the program, by converting the state C_0 to C_1.

Figure 3 summarizes the representative operational semantics rules, which are explained in detail next.

Place evaluation. For variable x, the rule (Var) first fetches the memory address l from the store Σ ($\Sigma(x) = l$), then fetches the value v from the heap Ψ ($\Psi(l) = v$) according to the memory address l. For array element access $p[x]$, the rule (Array) first evaluates the array index x to a value n, then evaluates the place p to an array value $\texttt{Array}[\ldots]$, thus the whole place $p[x]$ evaluates to the corresponding v_n. It should be noted that, to rule out buffer overflows, the array index n is checked against the array length k ($0 \leq n \leq k$). Other rules for evaluating places are similar.

Operand evaluation. Constant operands evaluate to themselves, as specified by the (Const) rule. The difference between move p and copy p is that the former will transfer ownership whereas the latter will not. However, from the operational semantics point of view, they both evaluate the value of the place p.

Right value evaluation. The notation $\&p$ creates an immutable reference to the place p without transferring ownership, and multiple immutable references to the place p can be created simultaneously. Furthermore, the notation $\&\texttt{mut}\ p$ creates a mutable reference to the place p, where aliasing is not allowed, as this reference may mutate the value of p. As a result, there can be at most one mutable reference to the place p.

Statement evaluation. The operational semantics for statements and terminators may change program states, by writing memories or invoking functions. Most rules are straightforward thus deserve no further explanations. In the panic call rule, since there is no place to store the returned value and the id of the basic block to transfer control to after executing the panic function, thus there are no relevant parameters in the call stack information, and we use the notation _ to represent a missing value.

IV. C to CRust Translation

This section presents the translation from C to CRust, by formally defining compilation rules.

A. Formalizing C

Figure 4 presents the syntax for a subset of C, which is used to specify the rules for translation to CRust. As our goal is to present the translation rules rigorously, we have included, in this syntax, key features of C, 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.

$$\begin{array}{llll}
\text{Type} & \tau & ::= & \texttt{bool} \mid \texttt{int} \mid \texttt{void} \\
\text{Value} & v & ::= & \texttt{true} \mid \texttt{false} \mid n \\
\text{Expression} & e & ::= & n \mid x \mid e + e \\
\text{Statement} & s & ::= & x = e \mid s_1; s_2 \mid \texttt{if}(e)\ s_1\ s_2 \\
& & \mid & \texttt{while}(e)\ s \mid \texttt{return}(e) \\
& & \mid & \texttt{return} \\
\text{Function} & f & ::= & t\ x(t_1\ x_1, \ldots, t_n\ x_n) \\
& & & \{t'_1\ y_1; \ldots; t'_m\ y_m; s; \}
\end{array}$$

Figure 4. A Representative Syntax for C

$$\frac{\Phi = \text{mapVar}([t_1\ x_1; \ldots; t_n\ x_n; t'_1\ y_1; \ldots; t'_m\ y_m]) \qquad \tau_{\text{ret}} = \text{mapTy}(t) \qquad \Phi \vdash s \rightsquigarrow (\vec{s'}, \vec{d})}{\vdash \left(\begin{array}{l} t\ x(t_1\ x_1, \ldots, t_n\ x_n) \\ \{t'_1\ y_1; \ldots; t'_m\ y_m; s; \} \end{array}\right) \rightsquigarrow \left(\begin{array}{l} \texttt{fn}\ x(x_1 : \tau_1, \ldots, x_n : \tau_n) \to \tau_{\text{ret}} \\ \{\texttt{let mut}\ y_1 : \tau'_1; \ldots; \texttt{let mut}\ y_m : \tau'_m; @\vec{d}; \vec{s'}; \} \end{array}\right)}$$

Figure 5. Rule for Compiling Functions

B. Compiling C to CRust

The rules for compiling C to CRust are specified by a set of judgments. We use the notation $\vec{s}$ for a list of CRustIR statements, and $\vec{e}$ for a list of expressions. We use the notation $[\,]$ for an empty list, and @ for list concatenation.

Compiling C functions. Figure 5 presents the rules for compiling a C function into a CRustIR function. 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 C type t to a CRustIR type τ:

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

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

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

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

$$\Phi \vdash s \rightsquigarrow (\vec{s'}, \vec{d}).$$

A C statement s is compiled to a tuple with two lists: $\vec{s'}$ is a list of CRustIR statements, and $\vec{d}$ is a list of variable declarations.

$$\boxed{\Phi \vdash s \rightsquigarrow (\vec{s'}, \vec{d})}$$

$$\frac{\Phi \vdash e \rightsquigarrow (\vec{s}, \vec{d}, y)}{\Phi \vdash x = e \rightsquigarrow (\vec{s}@[x = y], \vec{d})}$$

$$\frac{\Phi \vdash s_1 \rightsquigarrow (\vec{s'_1}, \vec{d_1}) \qquad \Phi \vdash s_2 \rightsquigarrow (\vec{s'_2}, \vec{d_2})}{\Phi \vdash s_1; s_2 \rightsquigarrow (\vec{s'_1}@\vec{s'_2}, \vec{d_1}@\vec{d_2})}$$

$$\frac{\begin{array}{c}\Phi \vdash e \rightsquigarrow (\vec{s_e}, \vec{d_e}, x) \\ \Phi \vdash s_1 \rightsquigarrow (\vec{s'_1}, \vec{d_1}) \quad \Phi \vdash s_2 \rightsquigarrow (\vec{s'_2}, \vec{d_2})\end{array}}{\Phi \vdash \texttt{if}(e)s_1s_2 \rightsquigarrow \begin{array}{l}(\texttt{bb0} : \vec{s_e}; \\ \qquad \texttt{switch}(\texttt{move}\ x) \to \\ \qquad [1 : \to \texttt{bb1}, \texttt{default} \to \texttt{bb2}]; \\ \texttt{bb1} : \vec{s'_1}@[\texttt{goto} \to \texttt{bb3;}] \\ \texttt{bb2} : \vec{s'_2}@[\texttt{goto} \to \texttt{bb3;}] \\ \texttt{bb3} : () \\ , \vec{d_e}@\vec{d_1}@\vec{d_2})\end{array}}$$

$$\frac{\Phi \vdash e \rightsquigarrow (\vec{s_e}, \vec{d_e}, x) \quad \Phi \vdash s \rightsquigarrow (\vec{s'}, \vec{d})}{\Phi \vdash \texttt{while}(e)s \rightsquigarrow \begin{array}{l}(\texttt{bb0} : \vec{s_e}; \\ \qquad \texttt{switch}(\texttt{move}\ x) \to \\ \qquad [1 : \to \texttt{bb1}, \texttt{default} \to \texttt{bb2}]; \\ \texttt{bb1} : \vec{s'}@[\texttt{goto} \to \texttt{bb0;}] \\ \texttt{bb2} : () \\ , \vec{d_e}@\vec{d})\end{array}}$$

$$\frac{\Phi \vdash e \rightsquigarrow (\vec{s}, \vec{d}, x)}{\Phi \vdash \texttt{return}(e) \rightsquigarrow ([\texttt{return}\ x]@\vec{s}, \vec{d})}$$

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

Figure 6. Rules for Compiling Statements

Most rules are straightforward. As an example, to compile "if(e) s_1s_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 "while(e) s", after the statement s has been compiled, the control will jump back to block BB0.

Compiling expressions. Figure 7 presents rules for compiling expressions, which are formalized by the following judgment:

$$\Phi \vdash e \rightsquigarrow (\vec{s}, \vec{d}, x)$$

An expression e is compiled into a three-element tuple: a list of CRustIR statements $\vec{s}$, a list variable declarations $\vec{d}$, and a variable x (representing the value of the expression e). Compilation rules are straightforward. As an example, to compile the addition $e_1 + e_2$, we first compile the two sub-expressions e_1 and e_2, to obtain $(\vec{s_1}, \vec{d_1}, x_1)$ and $(\vec{s_2}, \vec{d_2}, x_2)$, respectively. Then, we generate a new variable x as well as a new declaration $x : \texttt{int}$ to hold the addition of x_1 and x_2.

$$\boxed{\Phi \vdash e \rightsquigarrow (\vec{s}, \vec{d}, x)}$$

$$\frac{}{\Phi \vdash n \rightsquigarrow ([x = \texttt{const}\ n], [x : \texttt{int}], x)}$$

$$\frac{\Phi \vdash e_1 \rightsquigarrow (\vec{s_1}, \vec{d_1}, x_1) \qquad \Phi \vdash e_2 \rightsquigarrow (\vec{s_2}, \vec{d_2}, x_2)}{\Phi \vdash e_1 + e_2 \rightsquigarrow ([x = x_1 + x_2]@\vec{s_1}@\vec{s_2}, [x : \texttt{int}]@\vec{d_1}@\vec{d_2}, x)}$$

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

Figure 7. Rules for Compiling Expressions

V. Prototype Implementation

To conduct the evaluation, we have implemented a prototype for CRust, which consists of two main components: (1) a converter for translating both Rust and C code to CRustIR; and (2) a portion of prior program analysis algorithms and tools to CRustIR. To implement the converter, we leveraged the frontend of the Rust compiler `rustc` (latest version 1.63.0). Moreover, we have leveraged C2Rust [64], an off-the-shelf tool for legacy C code conversion, to translate the C code to Rust code then to CRustIR with the help of the Rust compiler front-end. For program analysis comparison, we ported existing algorithms and tools so that they can process CRustIR for cross-language program analysis. To be specific, we ported four state-of-the-art analysis tools: 1) `rustc` [72], the official static checker of Rust compiler; 2) Miri [73], an interpreter for MIR; 3) MirChecker [27], a vulnerability detection framework for Rust programs; and 4) Rudra [20], a vulnerability detection tool for `unsafe` code.

VI. Evaluation

In this section, we conduct experiments to evaluate CRust. We first introduce the datasets used for the evaluation (Section VI-B). Next, we evaluate the effectiveness (Section VI-C) and performance of CRust (Section VI-D).

A. 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.

B. Datasets

We used two datasets to conduct the evaluation: 1) micro-benchmarks; and 2) real-world CWE, containing a total of 131 (14+117) test cases.

TABLE II
EXPERIMENTAL RESULTS ON MICRO-BENCHMARKS

Test Case	Vulnerability Type	Source	CRUSTIR LOC	Conversion Time (s) / per line(ms)	Analysis Time (s) / per line(ms)	`rustc`	Miri	MirChecker	Rudra	CRUST (This work)
1	OOB	[49]	87	0.23 / 2.60	0.46 / 5.33	✘	✘	✘	✘	✔
2	stack overflow	[49]	559	0.22 / 0.40	0.56 / 1.00	✘	✘	✘	✘	✔
3	CFI violation	[49]	750	0.22 / 0.29	0.50 / 0.67	✘	✘	✘	✘	✔
4	meta data leaking	[49]	199	0.23 / 1.17	0.48 / 2.42	✘	✘	✘	✘	✔
5	UaF/DF	[49]	138	0.24 / 1.71	0.46 / 3.35	✘	✘	✘	✘	✔
6	Division by Zero	[27]	134	0.21 / 1.56	0.52 / 3.84	✘	✘	✘	✘	✔
7	OOB	[27]	162	0.23 / 1.39	0.43 / 2.68	✘	✘	✘	✘	✔
8	OOB	[27]	223	0.21 / 0.94	0.42 / 1.90	✘	✘	✘	✘	✔
9	UaF/DF	[65]	309	0.24 / 0.76	0.48 / 1.56	✘	✘	✘	✘	✔
10	integer overflow	[66]	215	0.21 / 0.99	0.54 / 2.52	✘	✘	✘	✘	✔
11	UaF/DF	[67]	356	0.24 / 0.68	0.49 / 1.37	✘	✘	✘	✘	✔
12	UaF/DF	[68]	351	0.23 / 0.64	0.49 / 1.41	✘	✘	✘	✘	✔
13	format string attack	[69]	92	0.23 / 2.47	0.47 / 5.06	✘	✘	✘	✘	✔
14	buffer overflow	[70]	287	0.23 / 0.82	0.54 / 1.88	✘	✘	✘	✘	✔

Micro-benchmark. We manually constructed a micro-benchmark consisting of 14 test cases with diverse types of vulnerabilities (as presented by the second column of TABLE II), including Out-Of-Bounds (OOB), Use-after-Free (UaF), Double Free (DF), divide-by-zero, integer overflow, and so on. These test cases are collected from two sources (third column of TABLE II): 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 [71] is a set of widely used "Weaknesses in Software Written in C", with a total of 117 vulnerable programs. Evaluating CRUST on well-established vulnerability sets like CWE demonstrates the effectiveness of CRUST on real-world multilingual applications. To use CWE for the evaluation of CRUST, we added a Rust wrapper to each code example in CWE, turning them into multilingual Rust applications with C.

C. Effectiveness

To evaluate the effectiveness of CRUST, we first apply CRUST to micro-benchmarks. The last five columns in TABLE II present experimental results, which demonstrate that CRUST is superior to the other four state-of-the-art studies (tools) in that CRUST successfully detected all vulnerabilities in these benchmarks whereas the other four tools detected none. This experiment shows that CRUST is effective in detecting vulnerabilities in multilingual Rust applications.

To investigate the effectiveness of CRUST on real-world programs, we applied CRUST to the CWE, our second benchmark. As TABLE III 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 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, 58 were successfully detected by CRUST whereas 11 were not. Hence, the true positive (TP) is 84.1% ($\frac{Success}{Valid}$).

TABLE III
EXPERIMENTAL RESULTS ON REAL-WORLD CWE

Dataset	Total	Filtered	Valid	Success	Miss	TP
CWE	117	48	69	58	11	84.1%

To further investigate the root causes for the 11 failed cases, we performed a manual analysis for them. This analysis revealed two important reasons: 1) C features unsupported by Rust; or 2) limitations of static program analysis. Among all 11 failed cases, 1 contains a feature unsupported by Rust, namely returning the stack address of a variable of unknown size, whereas the other 10 were just missed by the analysis itself. For example, CRUST does not analyze C library functions such as `strcpy` (2 cases), as it does not establish the semantic models for C libraries. Although establishing precise models for C libraries will lead to more accurate analysis, this task is tedious and laborious. Other missed cases include concurrency errors such as race conditions (3 cases), logical errors (4 cases), and numerical errors (1 case). Given our threat model, these failed cases should not be deemed as a limitation of CRUST.

D. Performance

TABLE II (the 5th and 6th columns) presents the performance of CRUST, including: 1) time for converting the source code to CRustIR (Conversion Time); and 2) time for program analysis on CRustIR (Analysis Time). We ran each test case 100 rounds, then calculated the average. Experimental results demonstrated that CRust is efficient in detecting vulnerabilities in multilingual Rust applications: the time spent on code conversion into CRust 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.

VII. 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 complete CRustIR model. While our formal definitions of the syntax and semantics of the specification language CRustIR model the semantics of Rust honestly, we rely on the off-the-shelf Rust type checker to check the type-related properties. By formally defining and incorporating a rigorous type system into CRustIR, we can make some program analysis algorithms, such as pointer analysis, more precise. Based on the latest research progress in this direction (e.g., RustBelt [83] and Patina [80]), we are attempting to formally define Rust's type system. Specifically, we are drawing on RustBelt's insight of logical relations to formally define a type system for CRustIR.

More expressive memory model. CRust's memory model makes use of a unified heap to store all variables. While this design models the Rust semantics honestly, it diverges from C's memory model, in which some variables are stack-allocated. Thus, by using a more expressive memory model to describe the memory layout of C, we may detect more subtle vulnerabilities. In this direction, we are planning to study a more elaborative C memory model, by introducing new CRustIR instructions to model the impact of C code on the Rust heap [92].

More comprehensive types of vulnerabilities. Although CRust can effectively detect memory-related vulnerabilities in multilingual Rust applications, CRust can also be extended to process more types of vulnerabilities. Especially, extending CRust with concurrency vulnerability checking capabilities will make it more comprehensive. In this direction, we can leverage the latest research progress (e.g., deadlock detection [74], data race detection [56], type confusion detection [26]) for concurrency security. We leave it an important future work.

VIII. Related Work

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

A. Rust Security

In the past few years, there have been a lot of studies on Rust security which can be classified into four categories: empirical study, vulnerability detection, vulnerability prevention, and formal verification.

Empirical studies. Current empirical studies on Rust security focus on security vulnerabilities, `unsafe` Rust, and the automatic C-to-Rust conversion tool C2Rust. Qin et al. [12] conducted an empirical study of memory and concurrency security vulnerabilities in Rust applications. Xu et al. [10] conducted an in-depth study of 186 memory security-related CVEs and proposed a taxonomy. Astrauskas et al. [75] studied the use of `unsafe` in 31867 Rust crates and summarized the usage scenarios of `unsafe`. Emre et al. [76] empirically investigated the limitations of C2Rust, and proposed some improvements.

Vulnerability detection. Vulnerability detection research mainly uses two techniques: 1) program analysis; and 2) fuzzing, in which program analysis is the most commonly used technique. SafeDrop [18], Mirchecker [27], Rupair [19], Stuck-me-not [74] and Rudra [20] all perform vulnerability detection based on program analysis. Another common technique is fuzzing. RUSTY [77] and RULF [78] are fuzzing tools for Rust programs and Rust libraries, respectively. In addition, Dewey et al. [54] proposed a fuzzing method for the Rust compiler.

Vulnerability prevention. Research related to Rust vulnerability prevention can be classified into two categories: 1) privilege separation-based; and 2) program analysis-based. The first strategy focuses on isolating `unsafe` code or data through memory access control and sandbox. FC [14] and XRust [15] prevent vulnerabilities by dividing memory into different regions with separate access control. Sandcrust [16] uses the sandbox to encapsulate the FFI. In addition, RUSBOX [17] uses a combination of static program analysis and sandbox to prevent buffer overflow vulnerabilities in Rust. The second strategy makes use of program analysis to visualize important information, such as lifetime and ownership, to help developers eliminate potential errors [79].

Formal verification. Research related to formal verification can be divided into two main categories: 1) formal semantics; and 2) automated program verification. For the formal semantics of Rust, Patina [80] is the first Rust formal semantics, which was subsequently extended by Lamqadem et al. [81]. RustBelt [83] is the first to include `unsafe`, which was subsequently extended by Dang et al. [84]. Rust automated program verification studies have all adopted a similar approach: first, translate the Rust program into an intermediate representation of an existing verification tool, and then use the existing tool to verify the Rust program. RSMC [56] utilizes Smack [86] to verify memory and concurrency security in Rust. Both Rust2Viper [57] and Prusti [75] use Viper [87] to verify the functional correctness of Rust.

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.

B. Multilingual Applications Security.

There have been many studies on the security of multilingual applications. PolyCruise [88] is a technique that generates a dynamic information flow analysis (DIFA) for multilingual software. Mergendahl et al. [49] systematically analyzed the security of multilingual applications and constructed threat models across Rust and C. BridgeTaint [89] is a two-way dynamic taint tracking method that detects security issues in communication between native code and Web code. Li et al. [90] reviewed the research results of cross-language static analysis in the field of Android application security. Costanzo et al. [91] proposed a formal verification method that verifies the data flow security of software systems consisting of C and assembly end-to-end.

Our work differs from the above efforts in that our research addresses the security of multilingual applications across Rust and C. We also propose an effective unified program analysis framework across Rust and C.

IX. CONCLUSION

This paper presents CRUST, the first unified program analysis framework for analyzing multilingual Rust applications. At the core of CRUST are a formal definition of a specification language CRUSTIR, as well as rigorous translation rules from C to CRUSTIR on which program analyses can be performed. We have implemented a prototype system for CRUST and conducted extensive experiments. Experimental results show that CRUST can effectively detect common memory security vulnerabilities across Rust and C that are not detected by other state-of-the-art Rust analyses, with negligible overhead. This work represents a new step towards securing Rust, making the promise of a safe 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] Durumeric, Z., Li, F., Kasten, J., Amann, J., Beekman, J., Payer, M., ... & Halderman, J. A. "The matter of heartbleed." *Proceedings of the 2014 conference on internet measurement conference*. 2014, pp. 475-488.

[2] Saito, T., Watanabe, R., Kondo, S., Sugawara, S., & Yokoyama, M. "A survey of prevention/mitigation against memory corruption attacks." *2016 19th International Conference on Network-Based Information Systems (NBiS)*. IEEE, 2016, pp. 500-505.

[3] Matt Miller. 2019. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL.

[4] Chromium. 2022. Chromium Security. https://www.chromium.org/Home/chromium-security/memory-safety.

[5] Steve Klabnik and Carol Nichols. 2018. The Rust Programming Language. https://doc.rust-lang.org/stable/book/.

[6] Ownership. https://kaisery.github.io/trpl-zh-cn/ch04-00-understanding-ownership.html.

[7] Borrow. https://doc.rust-lang.org/rust-by-example/scope/borrow.html.

[8] References. https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html.

[9] lifetime. https://doc.rust-lang.org/rust-by-example/scope/lifetime.html.

[10] Xu, H., Chen, Z., Sun, M., Zhou, Y., & Lyu, M. R. "Memory-Safety Challenge Considered Solved? An In- Depth Study with All Rust CVEs." *ACM Transactions on Software Engineering and Methodology*. 2021, pp. 1-25.

[11] Unsafe. https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html.

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

[13] Evans, A. N., Campbell, B., & Soffa, M. L. "Is rust used safely by software developers?" *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 246-257.

[14] Almohri, H. M., & Evans, D. "Fidelius charm: Isolating unsafe rust code." *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*. 2018, pp. 248-255.

[15] Liu, P., Zhao, G., & Huang, J."Securing unsafe rust programs with XRust." *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 2020, pp. 234-245.

[16] Lamowski, B., Weinhold, C., Lackorzynski, A., & Härtig, H. "Sandcrust: Automatic sandboxing of unsafe components in rust." *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*. 2017, pp. 51-57.

[17] Ouyang, W., & Hua, B. "RusBox: Towards Efficient and Adaptive Sandboxing for Rust." *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2021, pp. 1-2.

[18] Cui, M., Chen, C., Xu, H., & Zhou, Y. "SafeDrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis." *arXiv preprint arXiv*:2103.15420. 2021.

[19] Hua, B., Ouyang, W., Jiang, C., Fan, Q. "Rupair: Towards Automatic Buffer Overflow Detection and Rectification for Rust." *Annual Computer Security Applications Conference*. 2021, pp. 812-823.

[20] Bae, Y., Kim, Y., Askar, A., Lim, J., & Kim, T. "Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale." *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 2021, pp. 84-99.

[21] Ning, P., & Qin, B. "Stuck-me-not: A deadlock detector on blockchain software in Rust." *Procedia Computer Science*, 177, 2020, pp. 599-604.

[22] Erdin, M., Astrauskas, V., & Poli, F. "Verification of Rust Generics, Typestates, and Traits." Doctoral dissertation, Master's thesis, ETH Zürich, 2019.

[23] Merigoux, D., Kiefer, F., & Bhargavan, K. "Hacspec: succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust." Doctoral dissertation, Inria, 2021.

[24] Denis, X., Jourdan, J. H., & Marché, C. "The Creusot Environment for the Deductive Verification of Rust Programs." Doctoral dissertation, Inria Saclay-Île de France, 2021.

[25] Matsushita, Y., Tsukada, T., & Kobayashi, N. "RustHorn: CHC-Based Verification for Rust Programs." *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 43(4), 2021, pp. 1-54.

[26] Switzer, J. F. "Preventing IPC-facilitated type confusion in Rust." Doctoral dissertation, Massachusetts Institute of Technology, 2020.

[27] Li, Z., Wang, J., Sun, M., & Lui, J. C. "MirChecker: Detecting Bugs in Rust Programs via Static Analysis." *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2021, pp. 2183-2196.

[28] How much Rust in Firefox? https://4e6.github.io/firefox-lang-stats/.

[29] Function features. https://doc.rust-lang.org/book/ch13-00-functional-features.html.

[30] Enums and Pattern Matching. https://doc.rust-lang.org/stable/book/ch06-00-enums.html.

[31] Type checking. https://rustc-dev-guide.rust-lang.org/type-checking.html.

[32] Technology-most-loved-and-wanted-languages. https://insights.stackoverflow.com/survey/2021#most-loved-dreaded-and-wanted-language-love-dread.

[33] Tock. 2019. Tock Embedded Operating System. https://www.tockos. org/

[34] Lankes, S., Breitbart, J., & Pickartz, S. "Exploring rust for unikernel development." *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*. 2019, pp. 8-15.

[35] Levy, A., Campbell, B., Ghena, B., Pannuto, P., Dutta, P., & Levis, P. "The case for writing a kernel in rust." *Proceedings of the 8th Asia-Pacific Workshop on Systems*. 2017, pp. 1-7.

[36] Light A. "Reenix:Implementing a unix-like operating system in rust." Undergraduate Honors Theses, Brown University, 2015.

[37] Sung, M., Olivier, P., Lankes, S., & Ravindran, B. "Binoy Ravindran:Intra-unikernel isolation with Intel memory protection keys." *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 2020, pp. 143-156.

[38] Servo. The Servo Browser Engine. https://servo.org/.

[39] TFS. https://github.com/redox-os/tfs.

[40] Miller, S., Zhang, K., Chen, M., Jennings, R., Chen, A., Zhuo, D., & Anderson, T. "High Velocity Kernel File Systems with Bento." *19th USENIX Conference on File and Storage Technologies*. 2021, pp. 65-79.

[41] TTstack. https://github.com/rustcc/TTstack.

[42] A standalone, event-driven TCP/IP stack:smoltcp. https://github.com/smoltcp-rs/smoltcp.2022.

[43] Tokio is an asynchronous runtime for the Rust programming language. https://tokio-cn.github.io/.2022.

[44] TiKV. https://github.com/tikv/tikv.

[45] Parity. https://github.com/paritytech/parity-ethereum.

[46] Catalin Cimpanu. 2019. Microsoft to explore using Rust. https://www.zdnet.com/article/microsoft-to-explore-using-rust.

[47] Rust in android platform. https://security.googleblog.com/2021/04/rust-in-android-platform.html.

[48] Rust in linux kernel. https://security.googleblog.com/2021/04/rust-in-linux-kernel.html.

[49] Mergendahl, S., Burow, N., & Okhravi, H. "Cross-Language Attacks." *Proceedings 2022 Network and Distributed System Security Symposium.* NDSS. 2022, pp. 1-17.

[50] CVE-2021-25904: A raw pointer is dereferenced, leading to a read of an arbitrary memory address. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-25904.

[51] Foreign Function Interface. https://doc.rust-lang.org/nomicon/ffi.html.

[52] McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C. V., Shafi, H., Shanbhogue, V., & Savagaonkar, U. R. "Innovative Instructions and Software Model for Isolated Execution." *Hasp@ isca*, 10(1). 2013.

[53] Kil, C., Jun, J., Bookholt, C., Xu, J., & Ning, P. "Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software." *2006 22nd Annual Computer Security Applications Conference*. 2006, pp. 339-348.

[54] Dewey, K., Roesch, J., & Hardekopf, B. "Fuzzing the Rust typechecker using CLP (T)." *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ACM)*. IEEE, 2015, pp. 482-493.

[55] Lindner, M., Fitinghoff, N., Eriksson, J., & Lindgren, P. "Verification of Safety Functions Implemented in Rust-a Symbolic Execution based approach." *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*. Vol. 1. IEEE, 2019, pp. 432-439.

[56] YAN, F., WANG, Q., ZHANG, L., & CHEN, Y. "RSMC: A Safety Model Checker for Concurrency and Memory Safety of Rust." *Wuhan University Journal of Natural Sciences*, 2. 2020.

[57] Hahn, F. "Rust2Viper: Building a static verifier for Rust." Master's thesis, ETH Zürich, 2016.

[58] Payet, É., Pearce, D. J., & Spoto, F. "On the Termination of Borrow Checking in Featherweight Rust." *NASA Formal Methods Symposium*. 2022. pp. 411-430.

[59] The MIR(Mid-level IR). https://rustc-dev-guide.rust-lang.org/mir/index.html.

[60] Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., ... & Bastien, J. F. "Bringing the web up to speed with webassembly." *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2017, pp. 185-200.

[61] The MIR type-check. https://rustc-dev-guide.rust-lang.org/borrow_check/type_check.html.

[62] The move semantics in Rust. https://doc.rust-lang.org/std/keyword.move.html.

[63] The copy semantics in Rust. https://doc.rust-lang.org/std/marker/trait.Copy.html.

[64] C2Rust. https://c2rust.com/.

[65] CVE-2019-16144: Uninitialized memory is used. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-16144.

[66] CVE-2017-1000430: Buffer overflow. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1000430.

[67] CVE-2019-15551: Double freehttps. //cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-15551.

[68] CVE-2019-16140: Use-after-free. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-16140.

[69] CVE-2019-15547: Format string issues. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-15547.

[70] CVE-2019-15548: Buffer overflow. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-15548.

[71] CWE VIEW: Weaknesses in Software Written in C. https://cwe.mitre.org/data/definitions/658.html.Accessed:2021-02-04.

[72] What is rustc? https://doc.rust-lang.org/rustc/what-is-rustc.html.

[73] Miri. https://github.com/rust-lang/miri.

[74] Ning, P., & Qin, B. "Stuck-me-not: A deadlock detector on blockchain software in Rust." *Procedia Computer Science*, 177, 2020, pp. 599-604.

[75] Astrauskas, V., Müller, P., Poli, F., & Summers, A. J. "Leveraging Rust types for modular specification and verification." *Proceedings of the ACM on Programming Languages 3.OOPSLA*, 2019, pp. 1-30.

[76] Emre, M., Schroeder, R., Dewey, K., & Hardekopf, B. "Translating C to Safer Rust–Extended Version." *Object-Oriented Programming, Sys-tems, Languages, and Applications*, 2021, pp. 1-29.

[77] Ashouri, M. "RUSTY: A Fuzzing Tool for Rust." *Annual Computer Security Applications Conference*, 2020.

[78] Jiang, J., Xu, H., & Zhou, Y. "RULF: Rust library fuzzing via API dependency graph traversal." *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 581-592.

[79] Zhang, Z., Qin, B., Chen, Y., Song, L., & Zhang, Y. "VRLifeTime–An IDE Tool to Avoid Concurrency and Memory Bugs in Rust." *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2020, pp. 2085-2087.

[80] Reed, E."Patina: A formalization of the Rust programming language." University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-02,264. 2015.

[81] LAMQADEM, A. A. "A Formalization of the Static Semantics of Rust." Corso di Laurea Magistrale in Informatica. 2019.

[82] Wang, F., Song, F., Zhang, M., Zhu, X., & Zhang, J. "Krust: A formal executable semantics of rust." *2018 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. IEEE, 2018, pp. 44-51.

[83] Jung, R., Jourdan, J. H., Krebbers, R., & Dreyer, D. "RustBelt: Securing the foundations of the Rust programming language." *Proceedings of the ACM on Programming Languages 2. POPL*. 2017, pp. 1-34.

[84] Dang, H. H., Jourdan, J. H., Kaiser, J. O., & Dreyer, D. "RustBelt meets relaxed memory." *Proceedings of the ACM on Programming Languages 4. POPL*. 2019, pp. 1-29.

[85] Jung, R., Dang, H. H., Kang, J., & Dreyer, D. "Stacked borrows: an aliasing model for Rust." *Proceedings of the ACM on Programming Languages 4. POPL*. 2019, pp. 1-32.

[86] SMACK is both a modular software verification toolchain and a self-contained software verifier. https://smackers.github.io/.

[87] Viper is a language and suite of tools developed at ETH Zurich. https://www.pm.inf.ethz.ch/research/viper.html.

[88] Li, W., Ming, J., Luo, X., & Cai, H. "PolyCruise: A Cross-Language Dynamic Information Flow Analysis." *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 2513-2530.

[89] Bai, J., Wang, W., Qin, Y., Zhang, S., Wang, J., & Pan, Y. "BridgeTaint: a bi-directional dynamic taint tracking method for JavaScript bridges in android hybrid applications." *IEEE Transactions on Information Forensics and Security* 14.3, 2018, pp. 677-692.

[90] Li, Z., & Feng, G. "Inter-Language Static Analysis for Android Application Security." *2020 IEEE 3rd International Conference on Information Systems and Computer Aided Education (ICISCAE)*, 2020, pp. 647-650.

[91] Costanzo, D., Shao, Z., & Gu, R. "End-to-end verification of information-flow security for C and assembly programs." *ACM SIGPLAN Notices* 51.6, 2016, pp. 648-664.

[92] Tan, G., & Morrisett, G. "ILEA: Inter-language analysis across Java and C." *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*. 2007, pp. 39-56.