Comprehensiveness, Automation and Lifecycle: A New Perspective for Rust Security

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

***Abstract*—Rust is an emerging programming language designed for secure system programming that provides both security guarantees and runtime efficiency and has been increasingly used to build software infrastructures such as OS kernels, web browsers, databases, and blockchains. To support arbitrary low-level programming and to provide more flexibility, Rust introduced the `unsafe` feature, which may lead to security issues such as memory or concurrency vulnerabilities. Although there have been a significant number of studies on Rust security utilizing diverse techniques such as program analysis, fuzzing, privilege separation, and formal verification, existing studies suffer from three problems: 1) they only partially solve specific security issues but lack comprehensiveness; 2) most of them require manual interventions or annotations thus are not automated; and 3) they only cover a specific phase instead of the full lifecycle.**

In this perspective paper, we first survey current research progress on Rust security from 5 aspects, namely, empirical studies, vulnerability prevention, vulnerability detection, vulnerability rectification, and formal verification, and note the limitations of current studies. Then, we point out key challenges for Rust security. Finally, we offer our vision of a Rust security infrastructure guided by three principles: *Comprehensiveness*, *Automation*, and *Lifecycle* (CAL). Our work intends to promote the Rust security studies by proposing new research challenges and future research directions.

Keywords*—*Perspective*, *Rust*, *Security

I. Introduction

Rust [1] is an emerging secure system-level programming language designed to address the security issues of low-level programming languages such as C/C++. Due to its successful combination of security and efficiency, Rust has gained popularity in recent years. According to a developer survey conducted by Stack Overflow in 2021 [2], Rust was rated as the "most popular programming language", with 86.98% of developers are using, or considering using, Rust. In the meanwhile, Rust has been successfully used to build infrastructure software, such as operating system kernels [3] [4] [5] [6] [7], web browsers [8], file systems [9] [10], cloud services [11], network protocol stacks [12], language runtimes [13], databases [14], and blockchains [15]. Rust is also becoming more widely used in industries, such as Microsoft [16], Google [17], and even Linux [18].

To guarantee both security and runtime efficiency, Rust introduced a group of novel language features, including *ownership* [19], *borrow* [20], *reference* [21], and *lifetime* [22]. Rust also provided a set of strict security rules for these language features, which are checked at compile time. These security features, as well as these security rules, eliminate two categories of vulnerabilities in Rust: 1) memory vulnerabilities, including dangling pointers, double free, and use-after-free, which are common in languages such as C/C++; and 2) concurrency vulnerabilities, including race conditions caused by mutable variable sharing between threads [23].

While Rust provided strong security guarantees, the overly strict restrictions of security rules also make it difficult, if not impossible, to develop low-level system code. To support arbitrary low-level programming and to provide more flexibility to developers, Rust introduced the `unsafe` [24] feature, which allows arbitrary `unsafe` operations by bypassing the Rust compiler's static security checking. The `unsafe` feature of Rust, as a loophole, is indispensable in building low-level software such as system libraries. For example, an empirical study [25] demonstrated that nearly 30% of libraries use the `unsafe` feature, and more than half of the library functions are not checked by the compiler due to the chain of calls between library functions.

Unfortunately, due to the insecure nature of `unsafe` Rust, improper usage of this feature can easily lead to vulnerabilities and form new attack surfaces. For example, *all* memory vulnerabilities are related to the `unsafe` feature [26] [27], and so do most concurrency vulnerabilities [27] [28]. Given the importance of Rust in building infrastructure software, even a single bug can defeat the security guarantee of Rust and lead to serious consequences. To this end, the Rust security study has become an important research topic.

In recent years, there have been a significant number of studies on Rust security. According to the research themes, existing studies can be classified into five categories: (1) empirical studies that investigate Rust vulnerabilities [26] [27] [28] and `unsafe` feature [25] [29] pragmatically. With empirical results, one can obtain an understanding of the fundamental security mechanisms of Rust, analyze the root causes and potential attack surface, and provide correct assumptions for subsequent vulnerability prevention and vulnerability detection studies; (2) vulnerability prevention, to reduce or prevent the occurrence of vulnerabilities, by utilizing privilege separation [30] [31] [32] [33] [34] and program analysis [35] [36] [37] [38]; (3) vulnerability detection, to effectively detect Rust program vulnerabilities statically or dynamically, by using data flow analysis [39] [40] [41] [42], fuzzing [43] [44], abstract interpretation [45], and pattern matching [46]; (4) vulnerability rectification, to rectify program automatically by using program analysis and constraint solving [47]; and (5) formal verification, to perform security verification and prove properties of Rust programs, by using formal semantics [48] [49] [50] [51], model checking [52] [53] [54], theorem proof [55] [56] [57], and symbolic execution [58] [59] [60] [61].

Despite the aforementioned research progress, however, there are still three significant challenges to building a unified Rust security infrastructure. First, current studies on Rust security are not comprehensive enough. For example, vulnerability detection studies usually only focus on a specific kind of vulnerability, such as buffer overflow [47], deadlock [41], and type confusion [42].

Second, current studies on Rust security cannot be fully automated. For example, vulnerability prevention techniques based on privilege separation usually require manually adding RPC invocations [30], and program verification usually requires user-supplied program specifications [55] or even partial rewriting of the program [54].

Finally, current studies on Rust security only cover a specific phase of the Rust lifecycle instead of the whole Rust development lifecycle. For example, Sandcrust [32] can only stop memory security vulnerabilities in the vulnerability prevention phase, while RustHorn [62] can only act in the formal verification phase. We argue that Rust security is a systematic engineering process that should be explored from a broader perspective. To summarize, to perform an exhaustive and insightful study of Rust security, we need a comprehensive, automated, and full lifecycle Rust security infrastructure.

In this perspective paper, we present a new take on Rust security infrastructure, which is founded on three principles: *Comprehensiveness*, *Automation*, and *Lifecycle* (CAL). The lifecycle that we adopt consists of four phases: vulnerability prevention, vulnerability detection, vulnerability rectification, and formal verification.

Vulnerability prevention. Vulnerability prevention is the first phase of the Rust security lifecycle and is critical to prevent security vulnerabilities and security attacks on Rust systems. It enhances the defense of a software system by introducing Rust-specific security mechanisms, such as sandbox, lifetime visualization, and memory isolation for `unsafe` Rust, during the development phase of a Rust software system to prevent or block the occurrence of vulnerabilities.

Vulnerability detection. Even with effective vulnerability prevention mechanisms, Rust security vulnerabilities may still occur. Therefore, vulnerability detection, as the second phase of the Rust security lifecycle, offers a fundamental measure and essential method for significantly improving the security of Rust. Since the Rust compiler already provides strict security checks for safe Rust, vulnerability detection in Rust should focus on `unsafe` Rust.

Vulnerability rectification. After a vulnerability is detected, rectification is indispensable for fixing the vulnerability timely. However, manual vulnerability rectification is not only time-consuming but also error-prone. Therefore, the third phase of the Rust security lifecycle is automated vulnerability rectification, which is vital for reducing software development costs and improving software security.

Formal verification. A program without detectable vulnerabilities does not mean that it implements the intended function or meets the required security specifications. Formal verification, based on mature mathematical techniques, rigorously proves the security of a Rust program or guarantees a Rust program satisfies expected design properties and security specifications. Therefore, formal verification is the last phase of the Rust security lifecycle, providing higher security guarantees for Rust software systems.

Contribution. This work stands for the first step towards proposing a new perspective for Rust security to offer a comprehensive, automated, and lifecycle security infrastructure. To summarize, this work makes the following contributions:

- **Comprehensive survey.** To properly frame and illustrate our vision, our paper systematically and in-depth surveys the current state of research and practice for Rust security.
- **Insights and suggestions.** We present insights and suggestions on current key challenges that a novel Rust security infrastructure should address.
- **Vision.** We offer our vision of a comprehensive, automated, and lifecycle Rust security infrastructure and outline an actionable research agenda.

Outline. The rest of this paper is organized as follows. Section II provides an overview of the status and progress of Rust security research and practice. Section III presents the key challenges of current studies. Section IV proposes our vision of a Rust security infrastructure, and Section V concludes.

II. State-of-the-Art

To adequately describe our vision of the future Rust security infrastructure, it is essential first to step back and survey the existing Rust security research. Therefore, in this section, we first conduct a systematic and in-depth survey of the Rust security research and then give an overview of the relevant research and tools and point out current limitations that must be overcome to enable our proposed vision.

A. Methodology

In order to systematically analyze and summarize the research in this area, we first collected and screened the published papers since 2011 (Rust was officially released to the public in 2011) according to the following steps: 1) using the Google Scholar search engine, as well as the ACM, IEEE, Springer, and other paper databases; 2) searching the keyword "Rust security" ; 3) searching all the papers from 2011 to the present; and 4) manually screening and reviewing the papers retrieved by the above steps. In the selection process, we focused on valuable and representative papers in the fields of programming language, software engineering, and information security.

As a result, we screened a total of **54** papers, mostly from high-quality journals and conferences, such as POPL, CCS, and ICSE. By understanding these studies in-depth, we classify existing research into five categories: empirical studies, vulnerability prevention, vulnerability detection, vulnerability rectification, and formal verification.

According to the above research directions, TABLE I gives the classification statistics of the published papers. By analyzing the data in the table, we can conclude that the highest

TABLE I
DISTRIBUTION OF RESEARCH DIRECTIONS IN RUST SECURITY

Research Direction	Number	Percentage
Empirical Studies	8	14.81%
Vulnerability Prevention	11	20.37%
Vulnerability Detection	10	18.52%
Vulnerability Rectification	1	1.85%
Formal Verification	24	44.45%

TABLE II
RUST SECURITY VULNERABILITIES CLASSIFICATION AND ROOT CAUSE

Vulnerability Category	Specific Vulnerabilities	Root Cause
Memory Security	Buffer Overflow Read Uninitialized Memory Invalid Free Use after Free Double Free Null Pointer Dereference Type Confusion	All memory security vulnerabilities are related to the use of `unsafe` Rust, including `unsafe` functions, FFIs, and `unsafe` traits.
Concurrency Security	Double Lock (●) Conflicting Lock (●) Forget Unlock (●) Channel Misuse (●) Data Race (○) Atomicity Violation (○) Order Vilation (○)	The main cause of concurrency security vulnerabilities is the misunderstanding of Rust ownership and lifetime rules.

percentage of formal verification research is 44.45%, while the percentages of empirical studies, vulnerability prevention, and vulnerability detection are 14.81%, 20.37%, and 18.52%, respectively. There is currently only one paper dealing with vulnerability rectification for Rust.

B. Empirical Studies

As Rust is a relatively novel security programming language, empirical studies lay the foundation for research in all phases of the Rust security lifecycle by understanding Rust's security mechanisms and analyzing the root causes of vulnerabilities and potential attack surfaces. Therefore, empirical studies are an important guide for building a comprehensive, automated, and full-lifecycle Rust security infrastructure. Existing empirical studies mainly focus on Rust security vulnerabilities and `unsafe` feature.

Existing empirical studies have revealed that security vulnerabilities still exist in real software systems built on Rust [26] [27] [28]. Based on the in-depth analysis and summary of existing studies, this paper classifies Rust security vulnerabilities into two categories: memory security vulnerabilities and concurrency security vulnerabilities. TABLE II presents the vulnerability categories, specific security vulnerabilities, and root cause analysis.

1) Memory Security Vulnerabilities. Existing empirical studies show that Rust's strict security mechanisms guarantee that programs built with safe Rust will not suffer memory security vulnerabilities [26] [27]. However, code blocks tagged with the `unsafe` keyword will bypass the compiler's security checks, leading to security vulnerabilities. Furthermore, since `unsafe` Rust code and safe Rust code are in the same process address space, `unsafe` code has full access to the entire process space, leading to a vulnerable point and attack surface for memory security. The causes of memory security vulnerabilities can be classified into four categories [26]: (1) automatic memory reclamation errors ; (2) the use of `unsafe` functions and FFIs; (3) the use of advanced features [63] of Rust, such as trait [64]; and (4) other common memory errors, such as arithmetic overflows and boundary checking issues.

2) Concurrency Security Vulnerabilities. Rust supports both message passing and shared memory mechanisms between threads. To guarantee concurrency security, Rust uses the same ownership model as in memory security to ensure the safe sharing of data between threads [23]. However, existing studies [28] [27] show that concurrency security vulnerabilities in Rust programs still exist and can be categorized into deadlock-related (●) and non-deadlock-related (○) vulnerabilities. The main cause of deadlock vulnerabilities is a lack of complete understanding of Rust's lifetime rules, while the main causes of non-deadlock vulnerabilities are failure to protect shared resources and message passing errors.

Considering Rust's application scenario of system-level programming and the design goal of runtime efficiency, introducing the `unsafe` feature is an inevitable choice for language design. An empirical study of the `unsafe` feature shows that although less than 30% of Rust libraries use the `unsafe` feature, more than half of the library functions cannot be statically checked by the Rust compiler due to the call chain between library functions [25]. The main reasons for using `unsafe` feature include interacting with other languages, implementing complex shared data structures, using `unsafe` concurrency feature, improving performance, and reusing existing code [29] [27].

Although existing empirical studies of Rust security have yielded many results, they still have limitations. First, the research datasets they use are small, which is prone to overfitting. For example, the dataset used by Xu et al. [26] contains only 186 Rust memory security vulnerabilities, while more than 300 Rust memory security vulnerabilities have been reported [65] [66]. Second, their analysis of vulnerability generation mechanisms is not deep enough.

C. Vulnerability Prevention

According to the techniques employed, we classify existing Rust vulnerability prevention studies into two categories: privilege separation-based and program analysis-based. We analyze and compare the existing studies, and the results are shown in TABLE III.

1) Privilege Separation. Privilege separation typically separates code that may contain vulnerabilities and enables security enhancements by dividing various computing resources and entities into different groups and assigning different permissions. The technologies used in related studies can be classified into two categories: first, memory isolation is achieved by

TABLE III
SUMMARY AND ANALYSIS OF RUST VULNERABILITY PREVENTION STUDIES

Prevention Technology	Main Technique	Frameworks or Tools	Vulnerability	Research Progress
Privilege Separation	Memory Isolation	Fidelius Charm [30]	Memory	Although there have been many research results on vulnerability prevention based on privilege separation, these studies only address memory vulnerabilities and do not address protection against concurrency vulnerabilities.
		XRust [31]	Memory	
		Galeed [34]	Memory	
		PKRU-safe [67]	Memory	
	Sandbox	Sandcrust [32]	Memory	
		RUSBOX [33]	Memory	
Program Analysis	Lifetime Analysis	Dominik [35]	Memory, Concurrency	Related research prevents the occurrence of memory and concurrency vulnerabilities by analyzing the program's call stack, variable lifetime, and ownership.
		VRLifeTime [37]	Memory, Concurrency	
	Call Stack Analysis	Lindgren [38]	Memory, Concurrency	
	Ownership Analysis	RustViz [68]	Memory, Concurrency	

TABLE IV
SUMMARY AND ANALYSIS OF RUST VULNERABILITY DETECTION STUDIES

Detection Technology	Frameworks or Tools	Main Technique	Auxiliary Technique	Data Structure	Vulnerability	FP
Program Analysis	UnsafeFencer [46]	Pattern Matching	Runtime Detection	AST	Memory	○
	SafeDrop [39]	Data-flow Analysis	Taint Analysis	MIR	Memory	◐
	Rupair [47]	Data-flow Analysis	-	AST, MIR	Memory	●
	safeIPC [42]	Data-flow Analysis	Runtime Detection	MIR	Memory	○
	MirChecker [45]	Abstract Interpretation	Constraint Solve	MIR	Memory	●
	Rudra [40]	Data-flow Analysis	-	MIR, HIR	Memory	◐
	SyRust [69]	Taint Analysis	Program Aynthesis	MIR	Memory	○
	Njor [70]	Data-flow Analysis	Taint Analysis	MIR	Memory	○
	Stuck-me-not [41]	Data-flow Analysis	-	MIR	Concurrency	◐
Fuzzing	RUSTY [71]	Contribution-based Testing	Concolic Execution	Source Code	Memory	○
	Dewey [44]	Constraint Logic	-	Source Code	-	○
	RULF [43]	Fuzzing Target Generation	Program Synthesis	Source Code	-	○

dividing the memory space into different regions and providing access control to each region [30] [31] [34] [67]; second, the sandbox is used to isolate code or data [32] [33].

Although existing frameworks based on privilege separation can effectively prevent vulnerabilities, they still have two limitations: (1) they cannot be automated, requiring programmers to manually add calls to corresponding interfaces; and (2) the vulnerabilities they prevent are not comprehensive, and they only target memory security vulnerabilities.

2) Program Analysis. Program analysis is also an important technique used for vulnerability prevention. It analyzes the intermediate representations provided by the Rust compiler, such as MIR, LLVM IR, and AST, to obtain important information such as the lifetime and ownership of variables [35], the call graph of the program [38], and the location of implicit unlock, and then visualizes this information [36] [37]. Programmers can use this information to discover potential security threats during the development phase.

Although these tools can help programmers identify potential security threats, they still have three limitations: (1) poor usability, the representation of information they extract is complex and difficult for users to understand [35]; (2) low accuracy, limited by the analysis algorithms they use, some information will be lost during the analysis process [38]; and (3) incomprehensive, tools can only extract a certain kind of information, such as lifetime, ownership, and call graph.

D. *Vulnerability Detection*

Existing Rust vulnerability detection studies mainly use two types of techniques: program analysis and fuzzing. In this paper, we analyze, summarize and compare the existing studies, and the results are shown in TABLE IV, where ○ means that the work does not mention the specific false positive, ◐ means false positive is lower, and ● means the false positive is higher.

1) Program Analysis. Program analysis is the most dominant technique in Rust vulnerability detection. It is usually performed on intermediate representations provided by the Rust compiler, such as AST, HIR, and MIR, for data flow analysis [39] [47] [40] [41] [42], pattern matching [46], and abstract interpretation [45] to collect program properties and features. Program analysis is also often combined with auxiliary techniques, such as dynamic detection and constraint solving, for vulnerability detection.

Although Rust vulnerability detection studies based on program analysis have yielded significant results, they still have two limitations. First, they usually target only one specific security vulnerability, such as incorrect raw pointer dereference [46], incorrect memory release [39], buffer overflow [47], type confusion [42], and double lock [41]. Second, they cannot capture all program properties associated with vulnerabilities, which leads to high false positives. Moreover, manually screening for real vulnerabilities is a time-consuming

TABLE V
SUMMARY AND COMPARISON OF RUST AUTOMATION PROGRAM VERIFICATION STUDIES

Technology	Research	Tool	IR	Data Structure	Verifiable Properties	`unsafe`?	Annotation
Model Checking	CRUST [52]	CBMC	C	AST	Memory	✔	Filter
	RSMC [53]	Smack	Boogie	LLVM IR	Memory, Concurrency	✔	-
	Baranowski [54]	Smack	Boogie	LLVM IR	Functional Correctness	✔	Specification
	Kani [72]	CBMC	Goto-C	MIR	Functional Correctness	✔	-
	RustHorn [62]	SeaHorn	CHC	MIR	Functional Correctness	✘	-
Theorem Proof	Ullrich [55]	Lean	Lean	MIR	Functional Correctness	✘	Specification
	CREUSOT [56]	WHY3	WHY3	MIR	Functional Correctness	✘	Specification
	Denis [57]	F*	F*, low*	AST	Functional Correctness	✔	Specification
	AENEAS [73]	F*	F*	MIR	Functional Correctness	✘	Specification
	Rust-Stainless [74]	Stainless	Scala	HIR, THIR	Functional Correctness	✘	Specification
Symbolic Execution	Lindner [59]	KLEE	LLVM IR	LLVM IR	Memory, No Panic	✔	-
	Rust2Viper [60]	Viper	Silver	HIR	Functional Correctness	✔	Specification
	Prusti [61]	Viper	Viper	MIR	Functional Correctness	✘	Specification

and laborious task, which reduces detection efficiency.

2) Fuzzing. Fuzzing has been successfully used to detect vulnerabilities in Rust programs [71], Rust libraries [43], and the Rust compiler [44]. Related studies use value-based mutation, program synthesis, and constraint logic programming to generate a large number of test cases automatically and, at the same time, monitor the abnormal behavior of the detected program to find program vulnerabilities.

Although fuzzing is more effective than program analysis because it relies on runtime information and the vulnerabilities it finds must be reachable, the frameworks based on fuzzing still have three limitations. First, they require a lot of time and computational resources. Second, they cannot find logic errors that do not cause program crashes. Finally, they need to automatically generate a large number of test cases that can trigger more execution paths and have a vulnerability orientation. However, Rust's complex type system poses a great challenge to generate well-typed and valid test cases automatically.

E. Vulnerability Rectification

The rectification of vulnerabilities is the third phase of the Rust security lifecycle. Since the cost of vulnerability rectification is a very large part of modern software development, automated rectification of vulnerabilities is essential for software quality assurance. However, there are few automatic rectification theories and techniques for common Rust vulnerabilities, only Rupair [47] has proposed an automatic rectification technique for IO2BO vulnerabilities.

F. Formal Verification

Formal verification is the most active direction in Rust security research. For programs that have been detected and rectified for vulnerabilities, formal verification can further rigorously prove their security and functional correctness. We classify formal verification studies into two categories: 1) studies on the formal semantics of Rust; 2) studies on automated program verification of Rust.

1) Formal Semantics. The studies of Rust's formal semantics use mathematical tools to precisely define the semantics of Rust, providing support for the studies of Rust's expressiveness, reliability, and completeness. Therefore, Rust's formal semantics research is an essential foundation for program verification. Although there have been many studies on Rust's formal semantics [75] [49] [76] [48] [50] [77] [51] [78] [79] [80] and they use different methods and tools, they all adopt a similar technique, i.e., first defining the formal semantics for a subset of Rust, then explicitly modeling the Rust program based on that semantics, and finally completing the verification of the Rust program using theorem provers.

Although studies on this topic have made significant progress, there are still persistent challenges. First, the semantic models proposed by existing studies only consider a subset of Rust and cannot be extended to all features of Rust. Second, existing studies still require a lot of manual transcription.

2) Automated Program Verification. The main goal of automated program verification studies is to prove that the program satisfies a specific specification at runtime based on theories related to program verification (e.g., Hoare logic [81]) and using automated verification tools. One of the main differences between automated program verification and formal semantics is that the primary goal of automated program verification is to automate the verification process, thus effectively reducing the cost and extending it to real large programs.

We analyze and summarize the existing studies and classify them into three categories: model checking, theorem proof, and symbolic execution according to verification techniques. As shown in TABLE V, we present a detailed comparison of representative studies in Rust automated program verification in terms of verification tools, intermediate languages, data structures, verifiable properties, support for the `unsafe` feature, and required user annotations.

Existing studies have adopted a similar approach, i.e., they all translate programs into intermediate representations supported by existing verification tools based on the intermediate representation provided by the Rust compiler and then use the existing verification tools to complete the verification. Although there have been a large number of Rust automated program verification studies, they have some limitations. First, they can only verify a subset of Rust. Second, verification tools are costly to maintain because they require custom compilers, so as Rust continues to evolve, the verification tools need to be constantly updated, and most of these tools have not been further maintained after they were proposed.

III. Challenges for Enabling CAL Rust Security

Despite the fact that a plethora of frameworks, techniques, and tools are already available for constructing a Rust security infrastructure, a comprehensive, automated, lifecycle Rust security infrastructure does not yet exist due to several factors. In this section, we describe a diverse set of open problems that are the most prominent obstacles to achieving a comprehensive, automated, and lifecycle Rust security infrastructure, derived from our extensive research, in-depth knowledge, and practical research experience in Rust security.

A. Fragmentation

Fragmentation is mainly manifested in two aspects: (1) not full lifecycle; (2) incomprehensive. First, existing studies all address only one phase of the Rust security lifecycle. For example, Fidelius Charm [30] can only work on the vulnerability prevention phase, while Rudra [40] can only work on the vulnerability detection phase. Second, the existing studies are all incomprehensive, and they all target only a specific type of vulnerability or are only valid for a subset of Rust. For instance, the formal verification tool AENEAS [73] can only verify a subset of Rust for functional programming [82], while the vulnerability detection tool Stuck-me-not [41] can only detect double lock vulnerabilities. Although there have been many research results related to Rust security, they are all fragmented, and there is currently no work dedicated to organically integrating these research results, which is a major challenge in building Rust security infrastructure.

B. Low Degree of Automation

Although automation can significantly reduce the cost of developing and maintaining Rust software, and many Rust security frameworks strive to achieve automation, a low degree of automation remains a challenge for Rust security studies. For example, XRust [31], a vulnerability prevention framework, requires users to manually insert security checks for memory access operations into the source code, while the automated program framework Rust2Viper [60] still requires users to add program specifications that need to be verified. The low degree of automation of these frameworks makes them difficult to use for large real-world Rust projects due to the significant manual effort needed.

C. Lack of Breadth and Depth of Empirical Studies

Although the results of the empirical studies are important guidance for the security strategies in each phase of the Rust security lifecycle, the existing empirical studies on Rust security generally lack breadth and depth, and have three main limitations. First, the research datasets they use are small, which tends to lead to overfitting problems. The mainstream Rust vulnerability datasets CVE [65] and RustSec [66] have reported more than 400 Rust-related vulnerabilities, while the dataset used by Yu et al. contains only 18 vulnerabilities [28]. Second, existing studies do not provide an in-depth analysis of the connection between the root causes of vulnerabilities and Rust's security mechanisms, nor has it further proposed the best security practices for Rust's security features. Finally, the existing studies do not further explore the impact of vulnerabilities on Rust programs.

D. Absence of a Unified Program Analysis Framework

Program analysis is the most common technique used in Rust security studies [31] [39] [47] [40] [41] [42], and a unified program analysis framework can not only provide the foundation for Rust security studies, but also provide the necessary prerequisites for the continuous accumulation and evolution of research results. However, existing studies have directly used one or two intermediate representations provided by the Rust compiler, such as AST [83], HIR [84], and MIR [85], and no unified Rust intermediate representation and program analysis framework have been established. A single intermediate representation cannot contain information about the type, data flow, and control flow of a program at the same time, which makes it difficult to design high-precision program analysis algorithms and establish a unified program analysis framework.

E. Shortage of Vulnerability Rectification Technology

Although there have been many studies on the automatic rectification of vulnerabilities in Java [86] or C [87], the rectification theories and techniques proposed by these studies cannot be directly applied to Rust for two types of reasons. First, the new features introduced by Rust, such as ownership and explicit lifetime, pose challenges for automatic vulnerability rectification. Second, the automatic rectification of Rust vulnerabilities involves the interaction of `unsafe` code and safe code. Currently, only Rupair [47] has studied the automatic rectification of IO2BO vulnerabilities in Rust. In the future, research on automated rectification for more Rust vulnerabilities will help improve the efficiency and reduce the cost of vulnerability rectification.

F. Lacking a Full Language Formal Verification Model

Establishing a formal verification model for the full language of Rust is important to fundamentally ensure Rust security and promote the application of Rust in the security domain. Existing research has shown that the formalization of a full language for functional languages is feasible [88]. However, existing studies have completed formal verification of only a subset of Rust, due to two challenges. First, Rust introduced many new language features, including ownership model and lifetime. Second, Rust is a multi-programming paradigm language that integrates imperative, functional, object-oriented, and generic programming. Building a formal model of these new features and multiple paradigms is a challenging and urgent direction.

IV. Vision: CAL Rust Security Infrastructure

To instantiate a CAL Rust security infrastructure, the challenges listed in Section III must be addressed. The Rust security infrastructure is based on three core principles: comprehensiveness, automation, and lifecycle. In this section, we propose an infrastructure for Rust security following CAL principles. To make this vision feasible, we also propose a research agenda.

A. The CAL Principles

The Rust security infrastructure should be based on three principles: comprehensiveness, automation, and lifecycle.

Comprehensiveness. The comprehensiveness of the Rust security infrastructure is reflected in three aspects: vulnerability types, language features, and boundary cases. First, the Rust security infrastructure should be able to handle as many vulnerability types as possible. Second, it should also cover the main language features of Rust, especially ownership, lifetime, and concurrency, which are strongly related to security. Finally, it needs to handle a variety of boundary cases, including the interactions of `unsafe` code and safe code and Rust's interactions with other languages via FFI.

Automation. To achieve the design goal of automation, this infrastructure should provide a range of tools and techniques to minimize manual efforts, including but not limited to automated code staking, automated exploit generation, automated program verification, and automated vulnerability rectification. By maximizing the degree of automation, the difficulty of building software systems with Rust can be reduced, further promoting the widespread application of Rust in various fields.

Lifecycle. Although no research can solve all the problems in the entire lifecycle of Rust security, there are many research results in different phases of the lifecycle. Therefore, we can integrate the current research results and address the outstanding obstacles to building a Rust security infrastructure that can cover the full lifecycle. The full lifecycle Rust security infrastructure can provide all-around security protection and strong security assurance to Rust software systems at software development, testing and verification, thus driving the growth and maturity of the Rust ecosystem.

B. Proposed Infrastructure

In this subsection, we propose a conceptual Rust security infrastructure that follows the CAL principles and covers the four phases of Rust security lifecycle.

Vulnerability Prevention. In the vulnerability prevention phase, this infrastructure will first provide visualization tools to explicitly mark information such as ownership, lifetime scope, and call graphs to help developers discover potential errors. Then, `unsafe` code and data are isolated by memory isolation and sandbox techniques to prevent `unsafe` Rust code from affecting safe Rust code, thus preventing memory security vulnerabilities from occurring.

Vulnerability Detection. In the vulnerability detection phase, this infrastructure offers two detection techniques: program analysis and fuzzing. Program analysis will satisfy the following three requirements: first, it can perform interprocedural analysis; second, it can across the boundary of `unsafe` Rust and safe Rust; and finally, it can across the boundary of FFI for cross-language analysis. In addition, fuzzing can use program synthesis [89] to generate well-typed Rust programs as test cases, and then generate vulnerability exploits automatically, reducing manual efforts and following the principle of automation.

Vulnerability Rectification. In the vulnerability rectification phase, we should deeply analyze the generation mechanism of common vulnerabilities, summarize the code patterns that generate vulnerabilities, design effective automatic rectification strategies for various types of vulnerabilities, and then integrate the various rectification strategies into the IDE which developers can easily use. In this way, the principles of comprehensiveness and automation are followed.

Formal Verification. In the formal verification phase, this infrastructure offers automated program verification tools to verify the rectified program formally, which requires three steps. First, it needs to build a formal model of the whole Rust language. Second, it needs to customize a compiler to translate the Rust program into an equivalent verified program based on this formal model. Third, it needs to use existing verification tools for verification, such as SMACK [90], CBMC [91], and Viper [92]. Such an automated verification framework follows the principles of comprehensiveness and automation.

C. Research Agenda

Based on the existing theories, frameworks, and tools that are available to developers, as well as the limitations and remaining open challenges in the field of Rust security, we firmly believe that our vision for comprehensive, automated, and lifecycle Rust security infrastructure offers an architecture that, if realized, will dramatically improve the security of Rust software system. However, there are still many components of this vision that are yet to be adequately explored. Therefore, to make our vision tractable, we offer an overview of a research agenda broken down into seven major topics.

1) Comprehensive and In-depth Empirical Studies. The results of the empirical studies are instructive for the security research of the Rust security lifecycle. Therefore, before building a CAL Rust infrastructure, comprehensive and in-depth empirical studies are needed, and the research datasets they use need to include all reported Rust security vulnerabilities, mainstream Rust-based software systems, and ecosystem-level usage and evolution of Rust language features. Then, based on the datasets, we analyze in-depth the mechanism of Rust's security vulnerabilities, the relationship between Rust's security vulnerabilities and language mechanisms, and the impact of Rust's security vulnerabilities on the actual software. The conclusions are then summarized and refined to form the best security practices for the Rust language features, which can guide developers in their use of Rust. Finally, we provide more valuable suggestions for Rust language design and application development.

2) A Unified Program Analysis Framework for Rust. This program analysis framework consists of two main components: a unified intermediate representation and program analysis algorithms. Among them, the unified intermediate representation should contain as many program properties as possible. One possible solution is to integrate the AST, HIR and MIR provided by the Rust compiler into a unified intermediate representation, such as the code property graphs of C and Web Assembly that integrate abstract syntax tree, control flow graph, and program dependency graph. In addition, analysis algorithms designed based on the unified intermediate representation can enable cross-language program analysis by adding a component that translates other languages to the unified intermediate representation. Recently, Rudra [40] successfully detected 264 new memory security vulnerabilities on the Rust ecosystem by performing inter-procedural analysis based on AST and MIR, which demonstrates the feasibility of our proposed method. An effective program analysis framework will play a supportive role for vulnerability prevention and vulnerability detection.

3) Effective Vulnerability Prevention Frameworks. Effective vulnerability prevention frameworks mainly include two types: visualization tools and security protection based on privilege separation. On one hand, the visualization tools use Rust's program analysis technology to explicitly mark the static and dynamic characteristics of programs. The security protection frameworks based on privilege separation, on the other hand, use techniques, such as memory allocators, sandbox, and memory protection keys (MPK), to distinguish `unsafe` Rust from safe Rust and prevent errors in `unsafe` Rust from affecting safe Rust. Although the new features, such as ownership and lifetime, provided by Rust present new challenges for Rust vulnerability prevention studies, there have been a significant amount of studies [67] [31] [33] [68] demonstrating the feasibility of our proposed approach.

4) Comprehensive Vulnerability Detection Frameworks. Vulnerability detection frameworks are mainly based on program analysis and fuzzing, and for these two types of research, an important future research direction is to combine them with deep learning techniques. For the combination of program analysis and deep learning, a unified program analysis framework can be used to mine the explicit features of Rust programs, while deep learning techniques can be used to mine the implicit features of programs, and then the two features can be combined to form a complementary, which can more powerful support for Rust vulnerability detection. For the combination of fuzzing and deep learning, we can first train a deep learning model using Rust programs, and then use the model to automatically generate test cases with high path coverage and vulnerability orientation, thus alleviating the common path explosion and blindness problems in fuzzing.

5) Novel Vulnerability Rectification Strategies. Rust automated vulnerability rectification studies mainly face two challenges. First, existing rectification strategies for other languages cannot be applied to Rust directly, due to the new security features introduced by Rust, so we need to propose novel rectification strategies for Rust features. Second, security vulnerabilities in Rust can involve interactions between `unsafe` code and safe code, so the rectification strategies often need to consider the whole-program rather than the local code where the vulnerability occurs. The only existing study on automated vulnerability rectification for Rust is Rupair [47], which proposes an automated rectification strategy for IO2BO vulnerabilities and provides a valuable reference for automated rectification of other types of vulnerabilities.

6) A Rust Full-language Formal Verification Model. Rust's automated program verification tools all use the same technical path of translating Rust programs into the verification languages required by existing verifiers, such as Lean [93], F* [94], and Silver [92]. This translation process is essentially a conversion from the formal model of Rust to the formal model of the verification language, which is then implemented by a custom compiler. Therefore, building a formal verification model for the whole Rust language is necessary before implementing an automated program verification tool that can be used in the Rust ecosystem. This formal verification model needs to include as many Rust language features and programming paradigms supported by Rust as possible, which is a challenging but important research direction for Rust security.

7) Automated Program Verification Frameworks. We believe that there are two possible future research directions in the area of Rust automated program verification. The first is to propose generic verification techniques and tools for Rust to reduce the cost of updating and maintaining multiple verification tools at the same time. The second is to propose new Rust automated program verification frameworks based on existing verification techniques through a deep integration of multiple techniques, such as combining interpolation techniques with SMT and combining model checking with abstract interpretation. We can propose a configurable verification framework based on existing verification techniques, and integrate multiple verification techniques and solvers in the framework to verify Rust software systems, thus improving the performance and scalability of the verification framework.

V. Conclusion

Rust is an emerging programming language for safe system programming. To enable low-level programming and arbitrary unsafe operations, Rust introduced the `unsafe` feature, which can bypass compiler static checking, leading to security issues. Therefore, studies on Rust security are very important to guarantee the security of Rust-based infrastructure software. This perspective paper first surveys the current Rust security research in terms of empirical study, vulnerability prevention, vulnerability detection, vulnerability rectification, and formal verification. This paper not only studied the limitations of current studies, but also presented current key challenges. Finally, we offer our vision of a comprehensive, automated, and lifecycle Rust security infrastructure, to motivate future studies in this area.

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] The Rust Programming Language. https://doc.rust-lang.org/stable/book/

[2] Most loved languages survey. https://insights.stackoverflow.com/survey/2021.

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

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

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

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

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

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

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

[10] S. Miller, K. Zhang, D. Zhuo, and T. Anderson, "High Velocity Kernel File Systems with Bento," *Computing Research Repository (CoRR).* 2020.

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

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

[13] Tokio is an asynchronous runtime for Rust. https://tokio-cn.github.io

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

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

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

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

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

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

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

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

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

[23] Fearless Concurrency. https://doc.rust-lang.org/book/ch16-00-concurrency.html

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

[25] 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 (ICSE)*, pp. 246-257, 2020.

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

[27] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang, "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.

[28] Z. Yu, L. Song, and Y. Zhang, "Fearless concurrency? understanding concurrent programming safety in real-world rust software," *arXiv preprint* arXiv:1902.01906 .2019.

[29] V. Astrauskas, C. Matheja, F. Poli, P. Müller, and A. J. Summers,"How do programmers use unsafe rust?" *Proceedings of the ACM on Programming Languages 4.OOPSLA*. pp. 1-27, 2020.

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

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

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

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

[34] E. E. Rivera, "Preserving Memory Safety in Safe Rust during Interactions with Unsafe Languages," Doctoral Dissertation, Massachusetts Institute of Technology, 2021.

[35] D. Dominik, "Visualization of Lifetime Constraints in Rust," Bachelor Thesis, ETH Zürich, 2018.

[36] D. Blaser, "Simple Explanation of Complex Lifetime Errors in Rust," Bachelor Thesis, ETH Zürich, 2019.

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

[38] P. Lindgren, N. Fitinghoff, and J. Aparicio, "Cargo-call-stack Static Call-stack Analysis for Rust," *2019 IEEE 17th International Conference on Industrial Informatics (INDIN),* 2019.

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

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

[41] P. Ning, and B. Qin, "Stuck-me-not: A deadlock detector on blockchain software in Rust," *Procedia Computer Science.* pp. 599-604, 2021.

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

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

[44] K, Dewey, J. Roesch, and B. Hardekopf, "Fuzzing the Rust typechecker using CLP (T)," *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 2015.

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

[46] Z. Huang, Y. J. Wang, and J. Liu, "Detecting unsafe raw pointer dereferencing behavior in rust," *IEICE TRANSACTIONS on Information and Systems.* pp. 2150-2153, 2018.

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

[48] R. Jung, J. H. Jourdan, R. Krebbers, and D. Dreyer, "RustBelt: Securing the foundations of the Rust programming language," *Proceedings of the ACM on Programming Languages,* vol. 2, no. POPL, pp. 1-34, Jan. 2018.

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

[50] H.-H. Dang, J.-H. Jourdan, J.-O. Kaiser, and D. Dreyer, "RustBelt meets relaxed memory," *Proceedings of the ACM on Programming Languages,* vol. 4, no. POPL, pp. 1-29, Jan. 2020.

[51] R. Jung, H. H. Dang, J. Kang, and D.Dreyer, "Stacked borrows: an aliasing model for Rust," *Proceedings of the ACM on Programming Languages,*vol. 4, no. POPL, pp. 1-32, Jan. 2020.

[52] J. Toman, S. Pernsteiner, and E. Torlak, "Crust: a bounded verifier for rust (N)," *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 2015.

[53] F. Yan, Q. Wang, L. ZHang, and Y. Chen, "RSMC: A Safety Model Checker for Concurrency and Memory Safety of Rust," *Wuhan University Journal of Natural Sciences 2*, 2020.

[54] Baranowski, Marek, S. He, and Z. Rakamarić, "Verifying Rust programs with SMACK," *International Symposium on Automated Technology for Verification and Analysis,* vol. 11138, pp. 528–535, 2018.

[55] S. Ullrich, "Simple verification of rust programs via functional purification," *Master's Thesis, Karlsruher Institut für Technologie (KIT)* . 2016.

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

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

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

[59] M. Lindner, J. Aparicius, and P. Lindgren, "No Panic! Verification of Rust Programs by Symbolic Execution," *2018 IEEE 16th International Conference on Industrial Informatics (INDIN).* pp. 108-114, 2018.

[60] Hahn, and Florian, "Rust2Viper: Building a static verifier for Rust," Master's Thesis, ETH Zürich, 2016.

[61] 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, pp. 1-30, October 2019.

[62] Y. Matsushita, T. Tsukada, and N. Kobayashi, "RustHorn: CHC-based Verification for Rust Programs," *ACM Transactions on Programming Languages and Systems,* vol. 43, pp 1-54, December 2021.

[63] Advanced Features. https://doc.rust-lang.org/book/ch19-00-advanced-features.html

[64] Traits: Defining Shared Behavior. https://doc.rust-lang.org/book/ch10-02-traits.html

[65] Rust CVE. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=rust

[66] A vulnerability database for the Rust ecosystem. https://rustsec.org/

[67] P. Kirth, M. Dickerson, S. Crane, P. Larsen, A. Dabrowski, D. Gens, and M. Franz, "PKRU-safe: automatically locking down the heap between safe and unsafe languages." *In Proceedings of the Seventeenth European Conference on Computer Systems.* pp. 132-148, 2022.

[68] M. Almeida, G. Cole, K. Du, G.Luo, S. Pan, Y. Pan, and C. Omar, "RustViz: Interactively Visualizing Ownership and Borrowing," *In 2022 IEEE Symposium on Visual Languages and Human-Centric Computing.* pp. 1-10, 2022.

[69] Y. Takashima, R. Martins, L. Jia, and C. S. Păsăreanu, "Syrust: automatic testing of rust libraries with semantic-aware program synthesis," *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 899-913, June 2021.

[70] E. J. Njor, and H. Gústafsson, "Static Taint Analysis in Rust," Master's Thesis, Aalborg University, 2021.

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

[72] A. VanHattum, D. Schwartz-Narbonne, N. Chong, and A. Sampson, "Verifying dynamic trait objects in Rust," *Proceedings of the ICSE-SEIP.* 2022.

[73] S. Ho, and J. Protzenko, "Aeneas: Rust verification by functional translation," *Proceedings of the ACM on Programming Languages.* pp. 711-741, 2022.

[74] Y. Bolliger, "Formal Verification of Rust with Stainless," Master's Thesis, Laboratory for Automated Informal Systems Reasoning and Analysis, 2021.

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

[76] F. Wang, F. Song, M. Zhang, X. Zhu, and J. Zhang, "Krust: A formal executable semantics of rust," *2018 International Symposium on Theoretical Aspects of Software Engineering (TASE).* 2018.

[77] Weiss, Aaron, Daniel Patterson, and Amal Ahmed. "Rust distilled: An expressive tower of languages." arXiv preprint arXiv:1806.02693 (2018).

[78] X. Denis, "Mastering Program Verification using Possession and Prophecies," *32 ème Journées Francophones des Langages Applicatifs*, pp. 174-189, 2021.

[79] J. Schilling, "Specifying and Verifying Sequences and Array Algorithms in a Rust Verifier," Master's Thesis, FAU Erlangen-Nürnberg, 2021.

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

[81] V. R. Pratt, "Semantical considerations on Floyd-Hoare logic," *17th Annual Symposium on Foundations of Computer Science (sfcs 1976).* 1976.

[82] Functional Language Features: Iterators and Closures. https://doc.rust-lang.org/book/ch13-00-functional-features.html

[83] Syntax and the AST. https://rustc-dev-guide.rust-lang.org/syntax-intro.html

[84] Rust HIR. https://rustc-dev-guide.rust-lang.org/hir.html

[85] Rust MIR. https://rustc-dev-guide.rust-lang.org/mir/index.html

[86] S. Sinha, H. Shah, C. Görg, S. Jiang, M. Kim, and M. J. Harrold, "Fault localization and repair for Java runtime exceptions." *Proceedings of the eighteenth international symposium on Software testing and analysis.* pp. 153-164, 2009.

[87] C. Zhang, T. Wang, T. Wei, Y. Chen, and W. Zou, "IntPatch: Automatically fix integer-overflow- to-buffer-overflow vulnerability at compile-time." *European Symposium on Research in Computer Security.* Springer, Berlin, Heidelberg, pp. 71-86, 2010.

[88] D. K. Lee, K. Crary, and R. Harper, "Towards a mechanized metatheory of Standard ML," *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* 2007.

[89] S. Gulwani, O. Polozov, and R. Singh, "Program synthesis," *Foundations and Trends® in Programming Languages,* 4.1-2 (2017): 1-119.

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

[91] CBMC is a Bounded Model Checker for C and C++ programs. https://www.cprover.org/cbmc/.

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

[93] Lean is a functional programming language and an interactive theorem prover. https://leanprover.github.io/.

[94] F* is a general-purpose functional programming language. https://www.fstar-lang.org/.