Detect Stack Overflow Bugs in Rust via Improved Fuzzing Technique

Zhiyong Ren
School of Computer Science
Fudan University, Shanghai, China
20210240038@fudan.edu.cn

Hui Xu
School of Computer Science
Fudan University, Shanghai, China
xuh@fudan.edu.cn

***Abstract*—Stack overflow has been a common memory vulnerability for a long time due to limited stack memory. Deep or infinite recursion serves as the main cause to exhaust the stack memory and crash the program. As a relatively new system programming language, Rust suffers from stack overflow problem inevitably. However, there is no relevant tool to detect those stack overflow bugs in Rust programming language. In this paper, we propose a novel approach using fuzz technique to trigger stack overflow bugs in Rust projects. We first build a call graph on Rust MIR (Middle Intermediate Representation). In the call graph, recursions appear as cycles lying in the SCCs (strong connect components). To find the entry APIs of those SCCs, we leverage Tarjan's algorithm to locate the SCCs and then reversely BFS (Breadth First Search) to search for the APIs. After that, we modify the underlying logic of AFL (American Fuzzing Loop) to trigger stack overflow bugs through fuzzing those dangerous APIs. Specifically, we add a function call time counter to accelerate the fuzzing process. We conduct our experiments on several existing Rust CVEs (Common Vulnerabilities and Exposures) related to stack overflow. Experiments show that our approach can trigger stack overflow bugs in a short time.**

***Index Terms*—fuzz, Rust, recursion, stack overflow**

I. Introduction

STack overflow is a well-known memory vulnerability for a long time. In software, *stack overflow* occurs when stack memory is exhausted. Essentially, stack is a linear data structure that follows the principle of LIFO (last in first out). It occupies a limited amount of virtual address space when program runs, e.g., default 8MB on Linux and 1MB on Windows. The size of the stack is determined at the start of the program and depends on many factors, including machine architecture, programming language and the amount of available memory. It grows from top memory address to lower memory address in memory space, which is totally opposite of heap memory allocation. When a program attempts to use more space than is available on the stack, the stack is said to *overflow*, typically leading to a program crash.

The program embraces a risk of *stack overflow* when it contains recursion. Unconditional recursive functions, like Figure 1, overflow the stack due to its infinite invocation. Besides, those conditional recursive functions like *fibonacci recursive function*, whose invocation depends on their context, can also trigger *stack overflow* errors. Hackers may construct a nested input maliciously according to the control flow of the program and make the functions recurring deeply to crash this program.

Recursion serves as a common programming skill and exists in many projects. Forbid programmers from using recursion seems unfeasible in real-world programming. We program concisely using recursion. Correspondingly, we have to scarify the performance of the program due to the cost of calling a function and tolerate the hidden risk of *stack overflow*. As an emerging programming language that promotes memory-safety features, Rust has attracted many developers in recent years. With no runtime and garbage collector, it empowers performance critical services, runs on embedded devices and easily integrates with other languages. Its rich type system and ownership model guarantee memory-safety and thread-safety features, which enables developers to eliminate many classes of bugs during compile-time. Nowadays, many developers use Rust to build their project due to its memory-safety benefits.

However, just like other programming languages, Rust suffers from *stack overflow* vulnerability inevitably. Dozens of Rust projects use recursion to implement their logic, e.g., yaml-rust[14], serde[15], ammonia[16]. Eight *stack overflow* related CVEs are found in Rustsec (The Rust Security Advisory Database). Due to its memory-safety promise, Rust has low tolerance for memory-safety problems. Unfortunately, there is no relevant tool to detect those stack overflow bugs in Rust projects.

In this paper, we detect those stack overflow bugs in Rust crates via an improved fuzz approach. We conduct our experiments on several Rust CVEs and successfully trigger stack overflow bugs in those CVEs within a short time.

We summarize the contributions of this paper as follows:

DOI reference number: 10.18293/SEKE23-122

- We leverage Rust type information on MIR to build our call graph and deal with some dynamic features.
- We design an algorithm to find all the entry APIs of SCCs in the call graph for constructing fuzz targets.
- We modify the underlying logic of AFL to accelerate fuzz process of triggering stack overflow bugs in Rust CVEs.

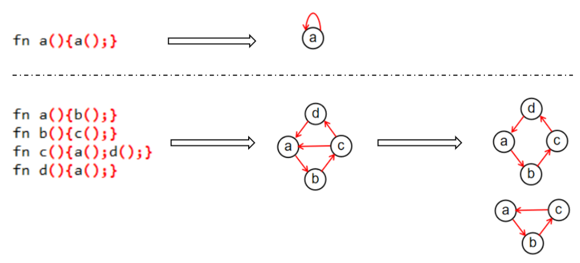
The rest of our paper is organized as follows. Section 2 presents our motivation and related work. After that, We introduce our approach in section 3. Section 4 demonstrates our evaluation experiments and Section 5 introduces limitations and future work. We conclude our paper in section 6.

II. MOTIVATION AND RELATED WORK

A. Motivation

As a static strongly typed language [5], Rust employs a stack to store some of its data. Especially, the Rust compiler puts those data types with fixed size, such as i32, u32, on the stack memory during compile-time, while allocates heap memory for those dynamic data types at run-time. When a function gets called in Rust, some stack memory gets allocated for its local variables, which is called a *stack frame*. *stack overflow* occurs when large amount of frames accumulates through recursive function calls.

Finding a recursive function in Rust crates seems troublesome due to the large amount of functions and complicated function call relations. Unfortunately, we lack a useful tool to automatically detect those unrevealed recursions in Rust crates. To our best knowledge, rustc (the Rust compiler) only provides warnings for those simple unconditional self-recursive functions. Nowadays, it still relies on human effort to locate those problematic recursive functions in Rust crates, which is labor-intensive. Moreover, developers may omit those possible cross-function recursions due to their complicated call relations like Figure 1. To tackle this problem, we propose a novel approach to trigger stack overflow bugs of those recursions which lie in SCCs of the call graph.



```
fn a(){a();}
```

```
fn a(){b();}
fn b(){c();}
fn c(){a();d();}
fn d(){a();}
```

Fig. 1. Recursive functions written in Rust and their recursive units.

B. Related Work

Rust has developed lots of useful artifacts recent years. Rudra [2] is a static analyzer to detect common undefined behaviors in Rust programs. SafeDrop [3] serves as a tool to detect memory deallocation bugs. RULF [4] helps to fuzz Rust libraries. Miri [22] provides an interpreter for Rust's mid-level intermediate representation. However, we still lack a tool to deal with *stack overflow* vulnerability in Rust crates.

So far, Rust developers have done some work on searching those self-recursive functions in Rust crates. The Rust compiler gives warnings on those simple unconditional self-recursive function. Wu[23] develops a clippy lint on HIR to detect those conditional self-recursive function. This lint simply compares the equality of caller def_id and callee def_id. Cargo-call-stack [21] develops a call graph on LLVM IR to analysis stack usage of each function, but it evades recursive functions.

To our knowledge, there is no formal work to test those recursions in Rust crates. Two open issues in the official rust-lang repository (issue 57965[18], 70727[17]) discuss those recursion problems but do not tackle it.

III. APPROACH

In graph theory, a strong connected component denotes a set of vertices and edges in which vertices can reach each other. Bascially, one strong connected component consists of serveral simple cycles. It can be decomposed into a certain number of independent cycles. Those cycles serve as recursions in real-world programs and have stack overflow risk. In the call graph, all recursions lie in the SCCs. So we choose to fuzz those SCCs in order to trigger stack overflow bugs. Our approach consists of three parts, including call graph construction, entry APIs searching and fuzzing. This section demonstrates these three parts in detail.

A. Rust Call Graph Construction

Call graph acts as a presentation of relations between functions in program and behaves as a high-level approximation of its run-time. As a normal static analysis approach, call graph presents an overview of the whole program and facilitates inter-procedure analysis.

Rust compiler (rustc) leverages a query system to enable demand-driven compilation. Incremental compilation implemented by rustc accelerates compiling process. During compilation, Rust source code goes through AST (Abstract Syntax Tree), HIR (High-level Intermediate Representation), MIR, LLVM IR [7] forms and finally turns into binary code. Rustc executes different operations on each compiling phase. For example, it does name resolution on AST while performs borrow check on MIR. We can choose

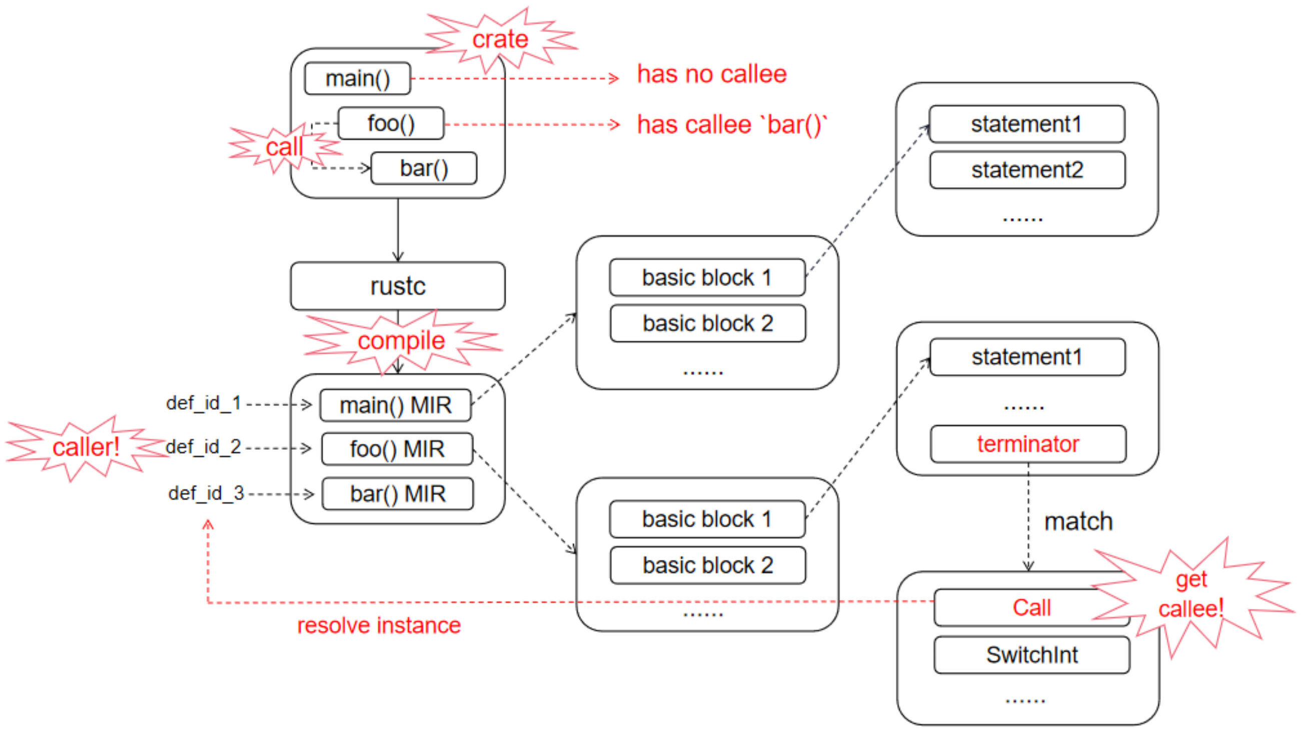


Fig. 2. Call graph construction.

to build our call graph on these four phases (AST, HIR, MIR and LLVM IR).

Due to the absence of Rust-unique type information, call graph built on AST, HIR or LLVM IR cannot resolve monomorphized features correctly, resulting in imprecise call edge. Therefore, we choose to build our call graph on Rust MIR and leverage type information to make call graph more accurate.

To deal with dynamic features such as dynamic dispatch in Rust, we collect all the types which implement the specific trait. For example, struct A, B, C implement trait `BoundTrait`. We cannot decide which function to call in compile time due to the absence of type information. So we take take all the types (A, B, C) into account to make our call graph sound.

To accelerate the construction process, we cast away functions which is calling into Rust *std* crate, e.g., println!.

We present the call graph construction process in Figure 2.

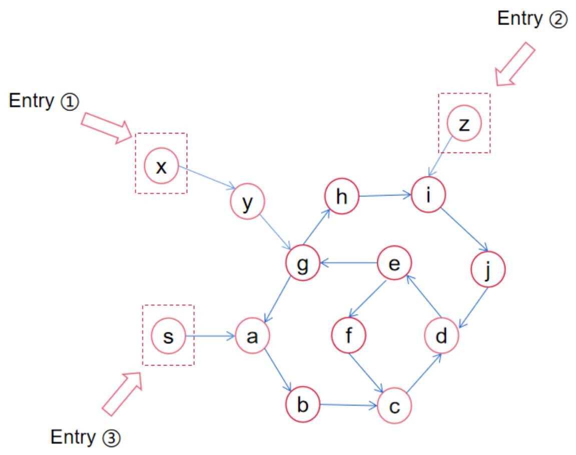


Fig. 3. Entry APIs of SCCs

B. Entry APIs Searching Algorithm

Cycles lie in strong connected components in call graph. To verify the risk of stack overflow bugs of the cycles, we need to find the entry points of all SCCs. We first leverage Tarjan's algorithm to find all the SCCs in the call graph. Then we reverse the direction of edges in the call graph. After that, we remove the edges in SCCs to avoid fruitless search and use BFS (breadth first search) to search for the entry APIs of each SCC. In Figure3, {a, b, c, d, e, f, g, h, i, j} forms an SCC and {x, s, z} denotes the entry APIs of this SCC.We demonstrate this algorithm in Algorithm1.

Algorithm 1 Search entry APIs of all SCCs

```
Input: $Adj$: adjacency list for a call graph
Output: $entryAPIs$: the entry API of each SCC
1: SCCs := Tarjan ($Adj$)   /* find SCCs */
2: reverse the edges of the call graph
3: for each scc in SCCs do
4:     remove the edges in the scc
5:     BFS to find entry APIs
6: end for
7: output those APIs
```

C. Fuzzing

To trigger *stack overflow* bug, we fuzz those APIs found by our searching algorithm. We choose AFL to do our fuzz job. To facilitate fuzzing process, we modified the underlying logic of AFL, including instrumentation and seed selection strategy. Specifically, we list the modifications as follow:

- **Instrumentation:** we extra instrument at the entry of each function and assign a unified ID `0` to these instrumentation. For other instrumentation lying in the branch, we generate a random ID from 1 to MAP_SIZE through a built-in function R(x). R(x) means generating a random number from 0 to x. So we set `x` with MAP_SIZE-1 to generate a random number from 0 to MAP_SIZE-1. Then we plus `1` to generate a number from 1 to MAP_SIZE. The whole expression is `R(MAP_SIZE-1)+1`. MAP_SIZE denotes the size of the trace_bits. So in trace_bits, trace_bits[0] denotes the times of function calls and trace_bits[1..MAP_SIZE] denotes the trigger times of each branch.
- **Seed Selection:** traditional seed selection strategy uses has_ new_ bits() function to determine whether the seed covers a new branch after one fuzzing loop. In addition to this, we take the function call times into consideration. That is, if a seed increases the maximum function call times after one fuzz loop, it will be added into the test queue for next mutation. Or it will be selected only if it increases the branch coverage. We use trace_bits[0] to record current function call times and a global variable to record maximum function call times.

Subsequently, AFL will carry out a large number of mutation on the selected seed and check whether it causes crash or finds new path. The main types of mutation includes bitflip, arithmetic, interest, dictionary, havoc, splice and so on. AFL will repeat the fuzzing loop until developers manually stop or timeout.

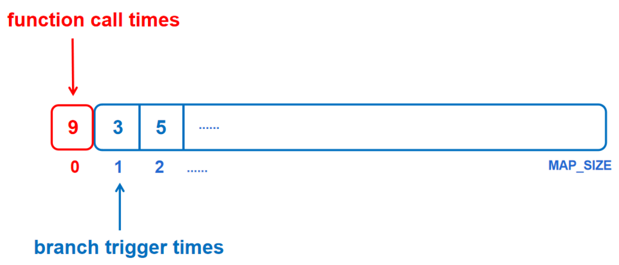


Fig. 4. trace_bits

IV. EXPERIMENTAL EVALUATION

This section presents our experiments on Rust crates. We evaluates our approach for three research questions.

- **RQ1:** How many dangerous APIs are there in our experiemntal crates?
- **RQ2:** Can our approach trigger stack overflow bugs successfully?
- **RQ3:** How much does our approach speed up the fuzzing process?

A. Experiment Setting

We collect eight Rust *stack overflow* related CVEs from RUSTSEC and download all related crates from GitHub. Recursion causes all of these CVEs. Users can crash the crates by constructing a nested input maliciously. Developers have located some of those problematic recursive functions in these CVEs and fixed them by limiting the recursion depth or changing the recursion into iterative form. Still, it remains questionable for some CVEs since it is not easy to locate the problematic recursions in dozens of functions with complicated call relations.

Before we test our approach on these crates, we make a new file named `rust-toolchain.toml` into the root directory of each crate to change the Rust toolchain into right version.

To trigger stack overflow bugs, we choose to fuzz those problematic CVEs. During the fuzzing process, we record the crash times every minutes. In order to speed up the fuzzing process, we use ulimit - s xxKB command to reduce the stack space into 512KB, 1024KB, 2048KB respectively. Moreover, we set the initial input with '{' . At the same time, we compare the unchanged AFL with improved AFL and observe the difference between these two experiments.

We conduct all our experiments on Dell-OptiPlex-7070 with 32GB memory and Intel Core i7-9700T 2.00GHz CPU.

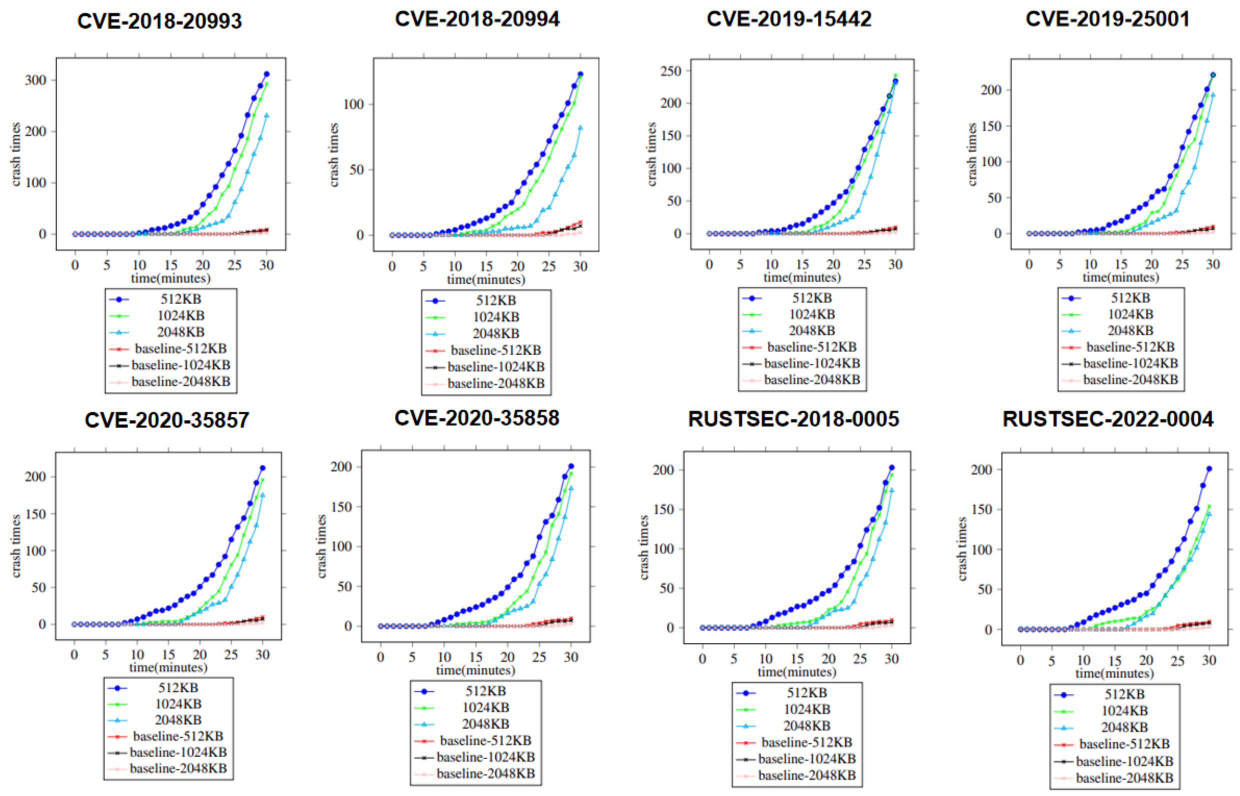


Fig. 5. experiment result on eight Rust CVEs

TABLE I
THE NUMBER OF DANGEROURS APIS IN CVE CRATES

CVE ID	Crate Name	number of APIs	
		dangerous	total
CVE-2019-25001	serde_cbor	27	451
CVE-2019-15442	markup5ever3	1	147
CVE-2020-35858	prost_build	1	124
CVE-2018-20994	trust-dns-proto	3	1390
CVE-2018-20993	yaml-rust	2	241
CVE-2020-35857	trust-dns-server	3	1390
RUSTSEC-2018-0005	serde_yaml	21	680
RUSTSEC-2022-0004	rustc-serialize	3	1504

TABLE II
THE NUMBER OF RECURSIONS IN CVE CRATES

CVE ID	Crate Name	number of recursions	
		self	cross-func
CVE-2019-25001	serde_cbor	5	0
CVE-2019-15442	markup5ever3	2	0
CVE-2020-35858	prost_build	2	0
CVE-2018-20994	trust-dns-proto	1	0
CVE-2018-20993	yaml-rust	1	13
CVE-2020-35857	trust-dns-server	0	1
RUSTSEC-2018-0005	serde_yaml	8	0
RUSTSEC-2022-0004	rustc-serialize	1	2

B. RQ1: How many dangerous APIs are there in our experimental crates?

In real-world Rust program, Rust developers always define those recursions using `private` keyword. They often offer an entry API to users at the top layer. This API is always set with `public`. We need to find these APIs to construct fuzz target.

We run our self-created algorithm to find the dangerous APIs of crate related to Rust stack overflow problem. Those dangerous API is defined as the entry APIs for all SCCs in the function call graph.

Experiment results in Table 1 show that each eight CVE related crate contains at least one dangerous API. In particular, serde_cbor shows the highest number of dangerous APIs, which is up to 27.

Those dangerous APIs lead to the recursions lying in SCCs. The recursions in Rust crate contain not only simple self-recursion but also complicated cross-function recursion. Table 2 demostrates the number of recursions in those CVE related Rust crate. All these recursions have the risk of stack overflow problem.

C. RQ2:Can our approach trigger stack overflow bugs successfully?

We have tested our approach on eight Rust *stack overflow* related CVEs. These crates contain library crates and binary crates. Figure 5 demonstrates our experiments on each CVE.

Results show that our approach has triggered stack overflow bugs. The blue line represents the experimental results at the stack space of 512KB, while the green line represents 1024KB and the blue line represents 2048KB. At the initial stage, the number of crashes is zero. After roughly eight minutes, the program crashes begin to occur, and the growth rate becomes faster and faster. The reason of the speed may be that the input generated by the mutation can cause the stack overflow bugs already, so each subsequent mutation can also cause stack overflow bugs. Moreover, the larger the stack space is, the longer time it takes to generate a crash.

We have carefully verified the correctness of each input generated by our approach. Specifically, we pass the input into the dangerous API, and it successfully triggers stack overflow bugs in the running process of the program.

D. RQ3: How much does our approach speed up the fuzzing process?

In Figure 5, the red, black and pink lines are the baselines which represent the unimproved AFL in the corresponding stack space. Compared with unmodified AFL, the modified AFL has a significant improvement in fuzzing speed. Take CVE-2018-20993 as an example, at the stack size of 512KB, the first crash occurs at minute 10 under the modified AFL while it occurs at minute 25 under the unmodified AFL. After five minutes, crash times grow into 16 while it only grows into 9 in baseline. At the time of 30 minute, the crash time reaches 312, which is far more than 9 in the baseline.

In other CVEs, it shows the same trend as it is in CVE-2018-20993. We see a clear gap of the experiment result between the improved AFL and baseline under both 512KB, 1024KB and 2048KB.

V. Conclusion

In this paper, we propose a novel approach to detect those stack overflow bugs in Rust crates. We first build a call graph on Rust MIR and use our algorithm to locate all entry APIs of SCCs in the call graph. Then we modify the instrumentation and seed selection strategy of AFL to accelerate the fuzz process of triggering stack overflow bugs. We conduct our experiments on Rust stack overflow related CVEs. Experiment results show that our approach successfully triggers stack overflow bugs under the stack size of 512KB, 1024KB, 2048KB. Our approach is the first attempt to tackle stack overflow problem in Rust programming language and has positive effect on further research.

References

[1] Evans, Ana Nora and Campbell, Bradford and Soffa, Mary Lou,Is Rust used safely by software developers?, IEEE/ACM 42nd International Conference on Software Engineering (ICSE), 246-257,2020.

[2] Bae, Yechan and Kim, Youngsuk and Askar, Ammar and Lim, Jungwon and Kim, Taesoo, *Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale*, Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. 84–99, 2021.

[3] Mohan Cui, Chengjun Chen, Hui Xu, and Yangfan Zhou. 2021. SafeDrop: De- tecting memory deallocation bugs of rust programs via static data-flow analysis.arXiv preprint arXiv:2103.15420, 2021.

[4] Jiang, Jianfeng and Xu, Hui and Zhou, Yangfan, RULF: Rust library fuzzing via API dependency graph traversal, 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), 581–592, 2021.

[5] Madsen, Ole Lehrmann and Magnusson, Boris and Molier-Pedersen, Birger ACM SIGPLAN Notices, 140-150. 1990.

[6] Taneja, Jubi and Liu, Zhengyang and Regehr, John, Testing static analyses for precision and soundness, Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization, 81-93, 2020.

[7] Lattner, Chris and Adve, Vikram, LLVM: A compilation framework for lifelong program analysis & transformation, International Symposium on Code Generation and Optimization, 2004.

[8] Wadler, Philip, The essence of functional programming, Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1-14, 1992.

[9] Krivine, Jean-Louis, About classical logic and imperative programming, Annals of mathematics and Artificial Intelligence.405-414,1996.

[10] Musser, David R and Stepanov, Alexander A, Generic programming 3rd ed, International Symposium on Symbolic and Algebraic Computation, 13-15, 1988.

[11] klabnik, Steve and Nichols, Carol.The Rust Programming Language.No Starch Press.2019.

[12] Jung, Ralf. Understanding and evolving the Rust programming language.Saarländische Universitäts-und Landesbibliothek.2020.

[13] Hoarem, Graydon.The rust programming language.http://www.rust-lang.org.2013.

[14] Chen YuHeng. 2021. yaml-rust: A pure rust YAML implementation. Github. https://github.com/chyh1990/yaml-rust

[15] David Tolnay. 2022. serde: Serialization framework for Rust. Github. https: //github.com/serde-rs/serde

[16] Laurent,iu Nicola. 2021. A whitelist-based HTML sanitization library: Repair and secure untrusted HTML. Github. https://github.com/rust-ammonia/ammonia

[17] Wojciech Danio.2020.Infinite recursion is not catched by the compiler.Github. https://github.com/rust-lang/rust/issues/70727

[18] Jonas Schievink. 2019. Make the unconditional_recursion lint work across function calls. Github. https://github.com/rust-lang/rust/issues/57965

[19] Tarjan, Robert, Depth-first search and linear graph algorithms, SIAM journal on computing, 146-160, SIAM.

[20] Johnson, Donald B, Finding all the elementary circuits of a directed graph, 3rd ed. SIAM Journal on Computing.77-84

[21] Lindgren, Per and Fitinghoff, Nils and Aparicio, Jorge Daly, Cargo-call-stack Static Call-stack Analysis for Rust, 2019 IEEE 17th International Conference on Industrial Informatics (INDIN),1169-1176, 2019.

[22] Cui, Mohan and Chen, Chengjun and Xu, Hui and Zhou, Yangfan, MIRI: An interpreter for Rust's mid-level intermediate representation, GitHub, 2022.

[23] AoXiang, Wu, clippy lint for dectecting conditional self-recursion, 2019 Github, 2021.