Rupair: Towards Automatic Buffer Overflow Detection and Rectification for Rust

Baojian Hua, Wanrong Ouyang, Chengman Jiang, Qiliang Fan, and Zhizhong Pan
School of Software Engineering, University of Science and Technology of China, China

ABSTRACT

Rust is an emerging programming language which aims to provide both safety guarantee and runtime efficiency, and has been used extensively in system programming scenarios. However, as Rust consists of an unsafe language subset `unsafe`, Rust programs are still vulnerable to severe security attacks which may defeat its safety guarantees. Existing studies on Rust security focus on the detection of vulnerabilities but seldom consider the bug fix issues. Meanwhile, it is often time-consuming and error-prone for Rust developers to understand and fix bugs manually, due to Rust's advanced language features. In this paper, we present Rupair, an automated rectification system, to detect and fix one sort of the most severe Rust vulnerabilities—buffer overflows, and to help developers release secure Rust projects. The key technical component of Rupair is a novel security oriented lightweight data-flow analysis algorithm, which makes use of Rust's two primary intermediate representations and works across the boundary of Rust's safe and `unsafe` sub-languages. To evaluate the effectiveness of Rupair, we first apply it to all 4 reported buffer overflow-related CVEs and vulnerabilities (as of June 20, 2021). Experiment results demonstrated that Rupair successfully detected and rectified all these CVEs. To testify the scalability of Rupair, we collected 36 open-source Rust projects from 8 different application domains, consisting of 5,108,432 lines of Rust source code, and applied Rupair on these projects. Experiment results showed that Rupair successfully identified 14 previously undiscovered buffer overflow vulnerabilities in these projects, and rectified all of them. Moreover, Rupair is efficient, only introduced 3.6% overhead to each rectified Rust program on average.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; • **Software and its engineering → Software notations and tools**.

KEYWORDS

Rust, buffer overflows, vulnerabilities, automatic program repair

ACM Reference Format:
Baojian Hua, Wanrong Ouyang, Chengman Jiang, Qiliang Fan, and Zhizhong Pan. 2021. Rupair: Towards Automatic Buffer Overflow Detection and Rectification for Rust. In *Annual Computer Security Applications Conference (ACSAC '21), December 6–10, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3485832.3485841

1 INTRODUCTION

Rust is a new programming language designed to help programmers write more secure and reliable system software, by employing a combination of compiler static and runtime checking. The main design philosophy for Rust is to inherit most features from C/C++, but to rule out safety issues in C/C++ [33], to achieve good performance without sacrificing safety guarantees. As a result, Rust has gained popularity in the past several years, and has been used successfully to build system software like operating system kernels [44, 47, 68], browser kernels [14], file systems [58], databases [15], cloud services [9], blockchains [4], and so on.

The Rust language can be divided into two sub-languages. The first one is the safe sub-language, consisting of a group of novel language features to support secure system programming. These safe features, including *ownership* [7], *borrow* and *move* [2], safe concurrency [77] etc., have been studied extensively. Existing research efforts, such as Patina [66], KRust [72], Rustbelt [38], Rust2Viper [34], RustHorn [55], etc., have successfully formalized (or mechanized) safety properties for this safe language subset of Rust.

In order to support arbitrary low-level operations and provide more flexibility to programmers, Rust introduced the unsafe sub-language with the `unsafe` [10] feature. Recent research [31] has shown that Rust's unsafe feature is used extensively by real-world Rust projects, about 50% of these projects used `unsafe` directly or in function call chains. Technically, this unsafe sub-language allows any operations that the Rust compiler cannot check for safety properties statically, thus may break the safety guarantees of the language. Due to the existence of this unsafe sub-language, severe threats happened on Rust programs [64] and a large number of vulnerabilities were reported. There have been significant research efforts to build practical safety analysis tools to help programmers detect bugs [54, 82].

However, all these aforementioned research efforts have severe limitations: they only detect the existence of bugs, instead of helping Rust developers fix the buggy code. Once a bug is detected, manually bug fixing for Rust programs is not only time-consuming, but also error-prone for several reasons. First, Rust's advanced programming features and their complex interactions pose challenges for developers, especially accounting for the average expertise of Rust developers is relatively low, as shown in Figure 1. Second, Rust is a relatively young language, and has undergone major changes in the past several releases, manually fixing programs for these incompatible releases is laborious. As a result, an automated vulnerability fix approached is expected to help Rust programmers develop secure programs. Otherwise, existing bug detection tools only benefit attackers to exploit buggy Rust programs.

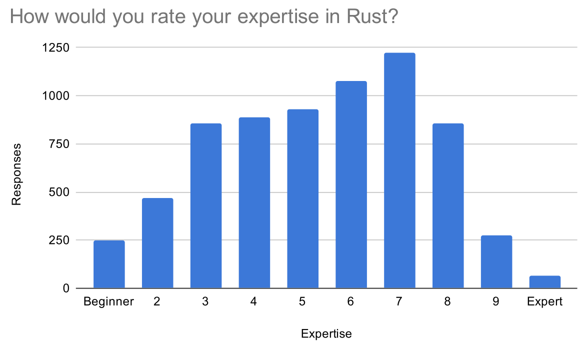

Figure 1: The overall expertise of Rust. (According to the official Rust developer survey [8], few programmers tend to claim expertise on Rust, and the peak is at 7.)

There have been a significant of studies to help programmers fix vulnerabilities automatically. The proposed techniques target different languages such as C [67], Java [19], Java bytecode [24], and recently, Ethereum Virtual Machine (EVM) bytecode [81]. However, the existing techniques cannot be applied to rectify vulnerabilities in Rust programs directly, because 1) the Rust language consists of some novel language features, such as ownerships [7] and explicit lifetimes [6], which do not exist in other languages; 2) most vulnerabilities in Rust arise from interactions between Rust's two safe and `unsafe` sub-languages [64], which does not appear in other languages and requires new techniques to handle. Thus, it's a challenge to develop automatic program repair techniques to fix vulnerabilities in Rust programs.

In this work, we present our first step towards an automatic rectification and protection infrastructure for buggy Rust programs. We designed and implemented an automated vulnerability detection and rectification system, Rupair, to fix one sort of the most severe vulnerabilities, buffer overflows, in Rust programs. The key technical insight behind Rupair's design is that vulnerabilities in Rust programs have certain insecure patterns, it is feasible to fix these bugs by identify and revise these patterns [64]. Rupair takes the following key steps to identify and rectify buffer overflow vulnerabilities: 1) Rupair identifies buffer overflow vulnerabilities with a novel data-flow analysis algorithm parameterized by overflow patterns, which scans across the safe and unsafe sub-languages boundaries, this step also addresses the aforementioned technical challenge; 2) Rupair confirms the existence of buffer overflows using Satisfiability Modulo Theory (SMT) solving techniques [20], which generates concrete counterexamples, this also reduces the false positives considerably; 3) Rupair conducts semantic-preserving program transformation to rectify the identified vulnerabilities; 4) for Rust programs that automatic rectification may have undesirable side effects or may break functionality consistence, Rupair constructs rectification suggestions and sends these suggestions back to developers, which may help them to rectify the bugs easier.

To evaluate Rupair, we first build datasets for experiments and analysis. First, we searched and collected all reported buffer overflow related CVEs and vulnerabilities [64], and have identified 4 of them (as of June 20, 2021). Second, we selected and collected 36 open source Rust projects, from 8 different domains, consisting of a total of 5,108,432 lines of Rust source code. The principal guiding the data selection process is to select as many domains as possible, and in each domain to select as many representative projects as possible. Though not a perfect metric, we measure "representative" by the number of downloads or stars of the corresponding projects on GitHub. As a result, the domains we selected cover a wide range of applications, and represent the typical usage scenarios of the Rust language.

With these datasets, we performed experiments and have obtained interesting insights and findings by analyzing the experiment results. First, to evaluate the effectiveness of Rupair, we applied it to all the 4 previously discovered vulnerabilities and CVEs. To testify whether the rectified Rust programs are secure, we simulated the attacker with both practical program analysis tools and real-world exploits. Experiment results showed that all Rust programs rectified by Rupair are bug-free and thwart all exploits.

Second, to testify the generality of Rupair, we utilized it on the collected 36 open source Rust projects. Experiment results showed that Rupair reported 29 buffer overflow vulnerabilities, among which 14 are real buffer overflow vulnerabilities (48.3%) (validated both by the Z3 solver and manual validation. To validate the normal functionalities of the rectified Rust projects, we conducted regression testing and found none of programs encountered execution inconsistency.

Finally, we conducted performance experiments, and found that it took Rupair 1853 milliseconds to analyze the 5,108,432 lines of Rust source code (2757 LOC per millisecond). Moreover, the experiment results showed that the overhead for the 36 rectified projects is 3.6% on average.

These experiments and results indicate that Rupair is both effective and efficient in identifying and rectifying buffer overflow vulnerabilities in real-world Rust projects.

To our knowledge, this work is the first automatic vulnerability rectification and protection system for Rust. To summarize, this work makes the following contributions.

- We present the first automated vulnerability identification and rectification approach for Rust programs. The core of this approach is a new security related data-flow analysis algorithm which scans across the safe and unsafe sub-language boundaries.
- We develop a fully automatic software tool prototype Rupair. This prototype is integrated with the above data-flow analysis algorithm, and leverages Satisfiability Modulo Theory (SMT) solving to generate concrete counterexamples which can trigger the vulnerabilities. For rectifications that may alter the semantics of the Rust programs, this prototype generates a detailed report which may help the Rust developers.
- We conduct systematic experiments, to testify the effectiveness, generality, and performance of this software prototype. Experiment results indicate this prototype is both effective and efficient in rectifying real-world buffer overflow bugs in Rust programs. In addition, this prototype reveals previously undiscovered vulnerabilities.

The rest of this paper is organized as follows. Section 2 presents the background and motivation for this work. Section 3 discusses, in detail, the design and implementation of the Rupair prototype. Section 4 presents the experiments we performed along with the datasets we used, and answers the research questions based on the experiment results. Section 6 discusses the related work, and Section 7 concludes.

2 BACKGROUND AND MOTIVATION

This section presents background and motivation for this work. Targeting on buffer overflow vulnerabilities in Rust programs, we have observed that a large portion of these vulnerabilities following common patterns. This fact indicates that these vulnerabilities can be identified and fixed through a unified approach. In the following, we first present examples to illustrate the common patterns. Then, we present the motivation for an automated rectification algorithm, which generates secure Rust source code by fixing these vulnerabilities. Thus, the security level of the entire Rust ecosystem is significantly improved.

2.1 Unsafe and Buffer Overflow Patterns

Although Rust is designed to be a safe system language, it contains an `unsafe` sub-language, which is the root cause for most vulnerabilities in Rust programs including buffer overflows [31, 64, 76]. In this section, we present the details.

The `unsafe` *Rust.* To provide better support for low-level system programming and enable Rust developers to write efficient programs, Rust introduced the `unsafe` language feature [10]. An `unsafe` code block may contain arbitrary statements that the Rust compiler cannot check safety statically, thus defeats the Rust language's strong guarantee of safety. Figure 2 presents typical usage

```
1   unsafe fn dangerous(){}
2   extern "C" { fn abs(input: i32) -> i32; }
3   unsafe trait Foo { ... }
4   unsafe impl Foo for i32 { ... }
5
6   fn f() {
7     let mut num = 5;
8     let r1 = &num as *const i32;
9     let r2 = &mut num as *mut i32;
10    let r3 = num as *const i32;
11    unsafe{
12      println!("r1={},␣r2={}", *r1, *r2);
13      *r3 = 6;
14      dangerous();
15      abs(-3);
16    }
17  }
```

Figure 2: The `unsafe` Usage Scenarios in Rust

of unsafe code: the code at line 1 declares an unsafe function, and the code spanning from line 11 to 16 is an unsafe code block.

This code is not complicated but illustrates all the 5 typical usage scenarios [10] of the `unsafe` feature in Rust: **1) raw pointer dereference**, the code at line 8 and 9 take addresses of the variable `num`, and create an immutable pointer `r1` and a mutable pointer `r2`, respectively. However, dereferencing raw pointers, such as `r1` or `r2`, is dangerous, as the Rust compiler cannot guarantee these pointers point to valid memory statically, thus, pointer dereferencing operations `*r1` or `*r2` should be placed in an `unsafe` block (line 12). Similarly, the variable `num` can be casted into a raw pointer `r3` directly (line 10), and the direct memory assignment is dangerous and should also be placed in the `unsafe` block (line 13). It should be noted that casting a reference into a raw pointer is safe (line 7 to 10), it's only dangerous to access memory through these raw pointers (line 12 and 13); **2) unsafe or foreign function invocations**, if a Rust function is marked `unsafe` explicitly (line 1), or is a foreign function (the `abs()` foreign C function at line 3), the invocations of such functions are dangerous and should be placed in an `unsafe` block (line 14 and 15); **3) unsafe trait**, a trait is unsafe if at least one function in it is unsafe (line 3, omitting the code in the trait `Foo` as it's unimportant), thus the concrete implementation of the trait is also unsafe (line 4). According to Rust specification [13], there are two more unsafe usage scenarios in Rust: static variable modification and union field access; however, these two features are rarely used in real-world Rust projects. The `unsafe` feature offers a challenge to the claim of Rust as a safe language. Recently, there have been a lot of research [29, 31, 49, 62, 64] to develop safety mechanisms to achieve Rust's vision of "pragmatic safety".

Buffer overflow patterns. Rust programs make extensive use of the `unsafe` feature to process buffers (i.e., vectors in Rust), because existing studies [64] have shown that buffer access in unsafe code is 4-5x faster than that in safe code, due to the absence of range checking. However, such unchecked buffer access can lead to severe overflows. Similar to previous research on bug taxonomy [79, 80], we categorize these bugs into 4 patterns, according to whether cause and effect are in safe or unsafe code: safe → safe, safe → unsafe, unsafe → unsafe and unsafe → safe. In the following, we use the symbols S and U to stand for safe and unsafe, respectively.

Figure 3 illustrates the four patterns of buffer overflows. The buffer `buf1` is both allocated and accessed in safe code, thus belongs to the pattern $S \rightarrow S$. Similarly, the buffer `buf2`, `buf3` and `buf4` belongs to the patterns $S \rightarrow U$, $U \rightarrow U$, and $U \rightarrow S$, respectively.

Buffer allocation arguments should also be checked for integer overflows [78], which may further trigger buffer overflows (i.e., the IO2BO bug pattern). For instance, the arguments of allocation at line 4 in Figure 3 may lead to integer overflows (it may allocate a vector of length 0), so buffer access at line 8 may lead to overflows, despite that the index `p+j-1` is always in bound.

Finally, it should be noted that as Rust compiler allocates all sorts of buffers in a single process address space, so any overflows in the `unsafe` block will corrupt the whole process memory [51].

2.2 Automated Rectification

We argue that the aforementioned buffer overflow pattern can be rectified through systematic program analysis and transformation. In this section, we propose an approach to automatically rectify

```
1   fn f(int i, int j, int k, int m) {
2     let mut buf1 = Vec::new();// S → S
3     buf1[i] = 10;
4     let buf2 = Vec::with_capacity(j * 4);// S → U
5     let buf4: Vec<i8>;
6     unsafe{
7       let p = buf2.as_ptr();
8       *(p + j - 1) = 20;
9       let mut buf3 = Vec::new();// U → U
10      buf3[k] = 30;
11      buf4 = Vec::with_capacity(k + m);// U → S
12    }
13    buf4[k + m] = 1;
14  }
```

Figure 3: Buffer Overflow Patterns. (The 4 patterns are written as $S \longrightarrow S$, $S \longrightarrow U$, $U \longrightarrow U$ and $U \longrightarrow S$, respectively.)

insecure Rust programs, by generating secure and correct Rust target programs.

Bug detection. Among all the 4 patterns, both $S \longrightarrow S$ and $U \longrightarrow S$ patterns are safe, as Rust always enforces range checking for buffer access in safe code. Hence, only patterns $S \longrightarrow U$ and $U \longrightarrow U$ can trigger buffer overflows. Furthermore, as the pattern $U \longrightarrow U$ is very similar to bug patterns in C/C++ and has been thoroughly studied [67, 78], so the only pattern of interest in this work is the pattern $S \longrightarrow U$. To detect this kind of bug pattern, we should design and implement a data-flow analysis algorithm, starting from all the use sites of some buffer `b` in an `unsafe` block, to calculate `b`'s definition sites in safe code (Section 3.3). Consider the program in Figure 3, the buffer `buf2` used at line 8 is defined at line 5. It should be noted that a full-fledged data-flow algorithm is required here, because the Rust programs being analyzed may contain complicated data dependency generally. For instance, the buffer `buf2` can be casted into a pointer `p` to access memory through it (line 7 and 8), here a standard alias data-flow analysis [17] is necessary.

Program rectification. After detecting bug candidates, Rupair applies the following techniques to rectify these bugs and generate secure and correct Rust programs. First, Rupair leverages SMT solver to valid the bugs, by generating counterexamples (Section 3.4). Second, data guards are generated and inserted for each buffer access in `unsafe` code blocks. Third, buffer allocation arguments are checked and transformed in a semantic-equivalent manner, to guarantee functionalities irrelevant to the insecure case are not affected (3.5); finally, Rupair rectifies the bugs and validates the rectified programs (Section 3.6 and 3.7).

3 SYSTEM DESIGN AND IMPLEMENTATION

In this section, we present the design and implementation of Rupair in detail. Designed to automatically fix insecure cases with typical insecure code patterns in Rust programs, Rupair takes as input the Rust source code and output secure Rust code without any buffer overflow vulnerabilities.

3.1 The Architecture

The architecture of the Rupair software prototype is given in Figure 4. Rupair consists of several key modules. The frontend takes as input the Rust source programs, and builds both abstract syntax tree (AST) and MIR intermediate representations. Next, the analyzer module scans the two intermediate representations, by using a data-flow analysis algorithm, to identify buffer overflow candidates. The solver module validates real buffer overflows from these candidates, by generating concrete counterexamples, and triggering overflows. The rectification module rectifies the buggy programs by a semantic-preserving program transformation. Next, the validation module validates the functionality equivalence between the rectified programs with the original ones, and generates the rectified programs, along with rectification reports to the Rust developers.

In the following sections, we present design and implementation details for each module.

3.2 The Frontend

The frontend of Rupair takes as input the Rust source files, and builds two intermediate representations: the Rust abstract syntax trees (AST) and the MIR. The AST is a tree representation of the source programs, especially, AST contains necessary source type informations for analysis in latter phases. The MIR is a control-flow graph (CFG) representation, in which each block is a sequence of statements. Blocks are connected by directed edges, which represents possible control transfers.

Rupair performs several rounds of static analysis on both AST and MIR in advance, to collect important information: 1) `unsafe` blocks, only functions that contain `unsafe` blocks will be further processed by latter modules; 2) variables' types, with a focus on variables having vector types; 3) source maps, mapping between a node in AST and the corresponding node in MIR. All these information will be used by the following phases, especially the data-flow analysis algorithm.

The design of Rupair differs from previous systems dramatically, in that it leveraged two IRs for Rust programs. The key reason for such a design will be discussed in Section 5.

3.3 Analysis Algorithm

Identifying precisely all buffer overflows statically is an undecidable problem, thus fixing all potential buffer overflows is intractable. We thus design an analysis algorithm, which is conservative in theory, but we have found it to be effective and efficient in practice. The key idea for this analysis algorithm is to: 1) identify all uses u of buffers in any `unsafe` code block B, which can be performed by a static program analysis, and 2) use a backward data-flow analysis to locate the definition d for each variable use u, whenever d satisfies some predefined buffer overflow detection criteria Q.

Algorithm 1 takes as input both a Rust program P and a set of overflow patterns Q, and calculates and returns a set of buffer overflows candidates, in R. First, this algorithm builds an abstract syntax tree A and a MIR M as aforementioned in Section 3.2. The algorithm visits each `unsafe` block u in the tree A, and calculates a set of live (thus used) variables in the block u by function `liveVars()`. The `liveVars()` function implements the standard liveness analysis

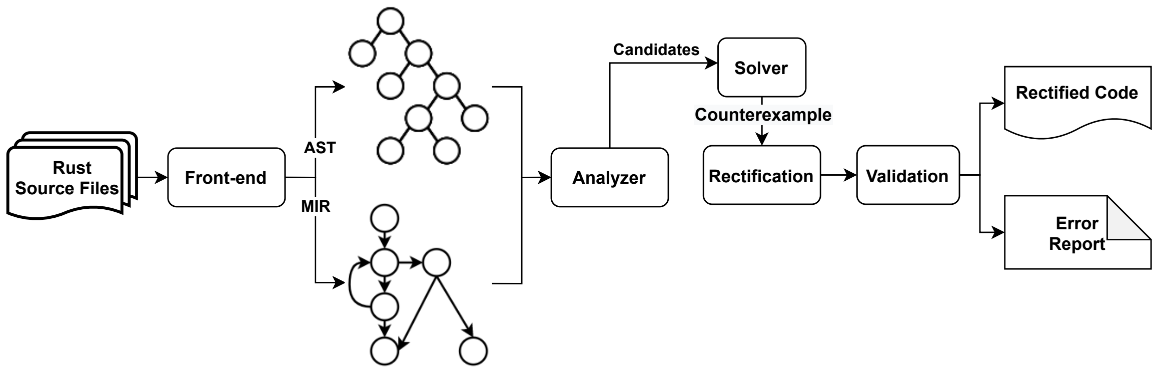

Figure 4: Rupair Architecture

Algorithm 1 : Calculating buffer overflow candidates

Input: P: The Rust program; Q: patterns used by heuristics
Output: A set of overflow candidates R

```
1:  procedure CAL-OVERFLOWS($P$, $Q$)
2:      $R = \phi$
3:      $A, M$ = buildAstMir($P$)
4:      for each unsafe block $u \in A$ do
5:          $vs$ = liveVars($M, u$)
6:          for each variable $v \in vs$ do
7:              if type($v$) == Vec<T> then
8:                  $ds$ = defSites($M, v$)
9:                  for each $d : v = alloc(e) \in ds$ do
10:                     if $d \in$ safe block of $A$ and $e \in Q$ then
11:                         $R \cup = d$
12:     return $R$
```

analysis algorithm as found in any compiler literature, thus deserve no further explanation. Next, this algorithm iterates each live variable v of any specific buffer type Vec<T> for some generic type parameter T, and calculates variable v's possible definition sites ds. The calculation for ds implements a variant of the standard reaching definition data-flow algorithm from program analysis. Next, the algorithm examines each definition site d for the variable v, and insert d into the buffer overflow candidates set R, when d is declared in the safe code block and belongs to the overflow detection patterns in Q. Finally, the algorithm returns the calculated set R containing all buffer overflow candidates.

Three important details in this algorithm deserve further explanations. First, this algorithm, like any static analysis algorithms, is conservative due to the incomplete nature of static analysis for runtime behaviors. For instance, consider the following Rust code fragment:

```
1 fn f(int x){
2   let mut buff = Vec::with_capacity(100);
3   unsafe{
4     if(x>=0)
5       buff[100] = 2;
6     else buff[20] = 3; } }
```

the variable buff lives at both line 5 and 6, thus the algorithm identifies the buffer allocated at line 2 as a candidate. However, as the execution of the if statement at line 4 is control-dependent on the variable x, the buffer overflow will only be triggered for input variable $x \geq 0$ (the notorious off-by-one bug). In despite of the conservativeness, we have observed, in our experiment (Section 4), that this algorithm is effective in practice with low false positives.

Second, this algorithm makes use of a pattern set Q, to specify possible forms of overflows. The pattern set Q is created in two ways. First, we systematically studied the Rust language specification [13], to identify all possible forms of overflows. For instance, the Rust language does not check overflows for arithmetic operations but allows wrapping around semantics by default [1], this fact indicates that all arithmetic operations, such as $e_1 \oplus e_2$ should be added to the set Q, where both e_1 and e_2 are expressions and $\oplus$ is any binary operators. Similarly, Rust does not check overflows for type coercions [3], but adopts the C language convention to truncate larger integers to smaller ones. Second, we systematically studied all discovered Rust bugs and CVEs [64], and identified buffer overflow-related bug patterns. Although this heuristic-based approach to create the pattern set Q is not technically complicated, it's laborious, taking 2 persons a month to finish. Furthermore, it should be noted that the key benefit of parameterizing this heuristics-base algorithm with the pattern set Q is that new overflow patterns can be added without changing the algorithm.

Finally, for a Rust function with M variables and N statements, the runtime complexity of this algorithm is $O(M * N)$. However, we have observed, during experiments, that this algorithm is very efficient, for most benchmarks, it runs in nearly linear time.

It should be noted that this algorithm is intra-procedural thus efficient. Although it's of no difficulty to scale this algorithm to a inter-procedural one, by creating a global call graph for the program being analyzed. However, doing so will slow down the analysis significantly, and we have found during experiments that this algorithm is effective in processing most programs.

3.4 Counterexample Generation

To identify real overflows from all the overflow candidates reported by the algorithm, we designed and implemented an automatic overflow validation module. The key insight for the design of this module is to generate counterexamples, by leveraging Satisfiability Modulo Theory (SMT) solvers.

Rupair's current implementation uses the Z3 solver [27] to generate counterexamples. The Z3 solver is selected, among many other solvers such as CVC4 [21], UCLID [42], etc., for several of its key advantages: 1) Z3 has complete support for linear arithmetic theory, which is used heavily by Rupair; 2) Z3 supports many convenient language bindings such as C/C++, Python, OCaml, and Java etc., Rupair makes use of its Python binding; 3) we have found that Z3 is efficient enough to process the constraints generated by Rupair.

There are three steps to generate counterexamples using Z3. First, Rupair generates constraints from Rust program's intermediate representations using Python binding; second, Rupair drives Z3 to process these constraints and generate concrete counterexamples; finally, Rupair feeds these counterexamples into the verifier module to justify the correctness by triggering overflows.

To testify the correctness of the counterexamples, Rupair makes of an instrumentation-based approach, in which Rupair explicitly sets variables values of the generated counterexamples and performs regressions. A promising approach to speed up the overflows triggering is program slicing [16, 22, 75] to slice the relevant program fragments automatically, but we have found the instrumentation approach is efficient enough in our experiments, thus leave the use of slicing a future work.

3.5 Rectification

After real buffer overflows are identified, Rupair rectifies the buggy programs by semantics-preserving program transformations. Rupair adopts two steps to finish the rectification: argument lifting and guard insertion.

Argument lifting. For the identified buffer overflow candidate programs, Rupair first lifts function call arguments by defining a transformation function $\mathcal{L}(\cdot)$ on an expression e: $\mathcal{L}(e) \Rightarrow e'$:

$$\begin{aligned} \mathcal{L}(f(e_1 \oplus e_2)) \Rightarrow & T_1\ x_1 = \mathcal{L}(e_1); \\ & T_2\ x_2 = \mathcal{L}(e_2); \\ & T_3\ y = x_1 \oplus' x_2; \\ & f(y); \end{aligned} \tag{1}$$

where the operator $\oplus$ stands for an arbitrary concrete operator such as $+$, $-$, $*$, $/$, etc.. Although this transformation looks straightforward, it's subtle to implement. First, the transformation function $\mathcal{L}(\cdot)$ is recursive, in that it transforms the sub-expression e_1 and e_2 recursively. Second, the transformation function $\mathcal{L}(\cdot)$ is type-preserving, it synthesizes types T_1, T_2 and T_3 for the newly generated variables x_1, x_2 and y, respectively. Rupair makes use of the type information on Rust program ASTs to synthesize these types. Finally, the transformation function $\mathcal{L}(\cdot)$ can be generalized to any function with n arguments $f(e_1, \dots, e_n)$, in which each argument $e_i, 1 \leq i \leq n$ can also be transformed by the equation (1).

Guard insertion. Rupair inserts specific data guards $\oplus'$ for arbitrary operator $\oplus$. A data guard is sequence of statements performing certain data validity checks. Rupair builds a secure library to perform secure operations using data guards. Rupair uses Rust's trait feature to define the secure operations, and the trait `SafeLib`

```
trait SafeLib<T>{
  fn checked_plus(&self, T y);
  fn checked_sub(&self, T y);
  ...
  fn on_flow(&self);
}
```

contains not only safe wrapper functions, such as `check_plus` etc., but also overflow handling functions, such as `on_flow()`. The key advantage of using a trait with a type parameter `T` is that this trait can be used with any data types that may have overflow bugs. With this secure library, Rupair performs guard insertion following a simple yet effective template-driven strategy as defined by:

$$\begin{aligned} x \oplus' y = x.&\texttt{checked\_} \oplus (y) \\ &\texttt{.on\_flow(Error :: new("Overflow for} \oplus \texttt{"))?;} \end{aligned} \tag{2}$$

By utilizing such data guards, Rupair checks the corresponding operations and precludes attacks.

It should be noted that as guard insertion replaces insecure operations by secure counterparts, and inserts extra security checking code into the rectified programs, this rectification incurs runtime overhead. However, experiment results (Section 4) demonstrate that this overhead is negligible for most test cases.

3.6 Validation

Automated program rectification may have undesirable side effects to change the programs' semantics or behaviors. In order to valid the normal functionalities of the rectified programs and to compare the semantic equivalence of the rectified programs with the original ones, Rupair makes use of two strategies to validate the rectified programs: regression and trace validation.

Regression. For the rectified programs, Rupair performs regression testing using the test cases distributed with each programs. Although it's well known that regression is incomplete, it is a well established and effective method for program testing, and Rupair's experiment results have shown this strategy is quite effective in practice.

Trace validation. Rupair also borrows the idea of trace validation from fuzzing [69] to check the equivalence between the rectified programs and the original ones. To be specific, Rupair records the execution traces by inserting random numbers into the head of each basic block in the programs' control-flow graphs. After running the programs, Rupair collects and compares the two execution traces for equivalence. Although this strategy is more complicated and requires more engineering efforts to implement, it's more powerful than the regression-based strategy as two programs with same outputs may take different execution traces.

Translation validation [60] is a more powerful approach to prove program equivalence, which is successful to prove the semantic equivalence of compiler optimizations. However, as program rectification described in this work does alter program semantics for the buggy programs, the translation validation technique can not be used in this scenario.

Table 1: A Micro Benchmark of Ground Truth in the Data Set

CVE or programs	App	Patching Time	Description	Pattern	Checked & Fixed?	Address Santinizer
CVE-2018-1000810	rust	2018-09	Creates a vector by repeating a slice n times.	Safe → Unsafe	Yes	No
RUSTSEC-2017-0004	base64	2017-05	Calculate the base64 encoded string size, including padding.	Safe → Unsafe	No	No
bfa13247	redox-ralloc	2018-06	Allocate fresh space that the space is allocated through a BRK call to the kernel.	Safe → Unsafe	Yes	No
fe905ed1	redox-relibc	2019-02	Allocate space.	Unsafe → Unsafe	Yes	No
array-mul	NA	NA	Array and num multiplication.	Safe → Unsafe	Yes	No
array-sum	NA	NA	Two array sum.	Safe → Unsafe	Yes	No
binary-search	NA	NA	Binary search algorithm.	Safe → Unsafe	Yes	No
find-duplicate	NA	NA	Find the number of duplicates in the array.	Safe → Unsafe	Yes	No
find-median	NA	NA	Looking for the median number of two positive sequences.	Safe → Unsafe	Yes	No
sorted-array-merge	NA	NA	Combine two ordered arrays.	Safe → Unsafe	Yes	No

3.7 Rectified Program Generation

After buffer overflows are identified and rectified, Rupair generates as outputs the rectified programs, along with rectification reports to the Rust developers.

Rectified program generation. As Rupair is designed to be a source-level rectification tool, it generates Rust source code for programs that are rectifiable.

Rectification report generation. Together with the rectified Rust programs, Rupair also generates a rectified report to the developers. For the successfully rectified programs, Rupair generates a summary containing the detailed description of the rectified program fragments. For buggy programs that automated rectification may have side effects, Rupair records them as "Unrectifiable" and reports suggestions to the developers for further manual inspection and rectification.

4 EXPERIMENTS AND RESULTS

In this section, we evaluate the effectiveness and efficiency of RUPAIR. Specifically, we aim to answer the following research questions:

RQ1: Effectiveness. As Rupair is proposed to automatically fix buffer overflow bugs in Rust programs, is it effective in fixing such vulnerabilities in Rust programs and real-world CVEs?

RQ2: Scalability. As Rupair is designed to automatically detect and fix buffer overflow vulnerabilities, how scalable is it and does it identify or fix previously undiscovered buffer overflow bugs in real-world Rust projects?

RQ3: Correctness. As Rupair is designed to automatically fix insecure Rust programs, how accurate is Rupair in fixing these bugs and assuring the functionality consistency between the rectified programs and the original ones?

RQ4: Cost. As Rupair is introduced to help Rust developers generate secure programs and may instrument the insecure source code, what's the performance of Rupair? Does Rupair introduce additional cost to the rectified programs?

4.1 Experimental Setup

We execute the latest Rust compiler version 1.51.0. All experiments and measurements are performed on a server with one 4 physical Intel i7 core (8 hyperthread) CPU and 8 GB of RAM running Ubuntu 18.04.

4.2 Data Sets

We select Rust projects to build data sets. There are three principals guiding our creation of data sets.

First, to testify the effectiveness and correctness of Rupair, we need to build a benchmark of ground truth. We built a micro benchmark of ground truth consisting of 10 buggy Rust programs as shown in Table 1. These test cases are created in two different ways: 1) we included all 4 known buggy programs from the public CVEs into our data set (the first 4 rows); and 2) we manually developed 6 other buggy Rust programs. It should be noted that the sole purpose of these micro benchmarks is to testify Rupair's effectiveness, not performance or cost, so the size of these benchmarks is irrelevant.

Second, to testify the scalability, performance and cost of Rupair, we aim to conduct experiments on real-world Rust projects. We systematically collected publicly available and open source Rust projects. In order to cover as many Rust usage scenarios as possible, we aim to include as many domains in our study as possible. As a result, we collected Rust projects from 8 different domains: databases, operating systems, gaming, image processing, cryptocurrency, security tools, system tools, and Web. These domains cover the most important usage scenarios of Rust. Furthermore, in each of above domains, we select as many representative Rust projects as possible. However, as with any open ecosystem, there exists a long-tail of projects in Rust that are small, largely unused or unmaintained. Therefore, we perform experiments on the more popular projects

Table 2: 36 Real-world Rust Projects in the Data Set

Domains	# Projects	LOC in Rust			Files of Rust			GitHub Stars		
		Avg.	Min	Max	Avg.	Min	Max	Avg.	Min	Max
Database	4	114238.0	7139	258379	316.5	35	661	4075.0	1100	9200
Operating System	3	71014.0	13396	174924	307.0	117	676	1810.7	432	2800
Gaming	4	57313.0	11410	182129	225.5	49	671	2587.0	948	6400
Image Processing	3	14171.7	9370	20976	52.0	37	66	1565.3	996	2400
Cryptocurrency	5	171778.2	2351	336798	583.0	6	1338	6520.0	3200	15800
Security tools	4	11621.2	2154	28460	73.5	9	174	1729.5	667	3900
System tools	7	7920.9	1223	22681	36.7	4	94	5171.4	2300	13000
Web	6	39989.6	14723	66384	144.8	68	225	22820.0	6800	75000

in our data set. As we download these Rust projects from both the central Rust repository and GitHub, we measure popularity by having the higher downloads or GitHub stars.

The selected domains and projects are presented in Table 2. For each of the 8 domains included, we give the numbers of selected projects in the corresponding domain, the sizes of these projects (measured by lines of source code), the numbers of Rust source files, and the GitHub stars.

In total, there are 36 projects, with 3 to 7 projects in each selected domains, respectively. These projects are selected based on their importance and popularity in the corresponding domain, according to the aforementioned data set selection criteria.

4.3 RQ1: Effectiveness

To answer **RQ1**, we first evaluated Rupair against the micro benchmarks in the data set (i.e., the ground truth). In total, Rupair successfully identified 9 buffer overflow vulnerabilities in 10 benchmarks, and Rupair successfully generates a rectified patch for each of the 9 benchmark (the 6th row in Table 1). The only program Rupair failed to analyze is RUSTSEC-2017-0004. A further investigation reveals that this program contains a function call, thus Rupair is unable to analyze due to its intra-procedural design decision (as discussed in Section 3.3). For such rare cases, Rupair constructs a report and sends to Rust developers for further manual inspection and rectification.

In order to verify whether Rupair has actually fixed the insecure code for each test case in this benchmark, we used the following strategies to conduct verification. First, we applied the state-of-the-art techniques, i.e., program analysis tools, to analyze these Rust programs and compare with Rupair.

To be specific, we used AddressSanitizer [11], a fast and widely-used memory error detector, to scan these benchmarks. As the last row in Table 1 shows, the AddressSanitizer failed to detect any vulnerabilities in these benchmarks. We further analyzed the results and investigated the root causes. The AddressSanitizer used instrumentation to insert specific range checking code to the programs being analyzed, and linked the programs with prebuilt shared libraries. However, as the prebuilt AddressSanitizer libraries only check a program against concrete inputs, thus it is unable to identify potential buffer overflow if the concrete inputs are in range. On the other hand, Rupair checks the array index symbolically using SMT solvers, it can identify potential overflows by constructing counterexamples.

Second, we extracted the semantics of the patch generated automatically by Rupair against the patch submitted for the CVE vulnerability or the original correct program that we manually constructed, and compared the program semantics to confirm the correctness of the patch generated by Rupair.

Finally, we evaluated whether the rectified programs by Rupair can defend against real-world attacks. To replay such attacks, we collected historical attack data for existing CVEs [5], and replayed them on the rectified Rust programs. Consequently, the experiment results showed that all the rectified programs defeat these attacks. This demonstrated that the Rupair can protect Rust programs against real-world attacks.

The above experiments demonstrate that Rupair is effective in identifying and rectifying buffer overflow bugs.

4.4 RQ2: Scalability

To answer **RQ2**, we applied Rupair to the benchmarks in our data set in Table 2, with 36 real-world Rust projects and a total of 5,108,432 lines of Rust code.

In Table 3, we present the projects for which RUPAIR reported buffer overflow vulnerabilities. In total, RUPAIR reported 29 buffer overflow warnings in 9 projects, by the Algorithm 1. Among these warnings, 14 were confirmed to be true buffer overflow vulnerabilities by the SMT solver Z3. We present the accuracy numbers A in the last row of Table 3, and A is calculated by

$$A = B/W,$$

where B and W stand for number of bugs and warnings, respectively. The accuracy is 48.3 on average, for all the programs.

4.5 RQ3: Correctness

To answer **RQ3**, we validate whether the functionalities of the rectified programs and the original ones are consistent. To conduct such validations, we use the test data distributed with each project to perform regressions. We executed each Rust program twice, separately on the rectified program and the original one, from the same execution state. Then, we compare the outputs from the two executions. If the outputs are the same, then the rectification is marked as "PASS"; otherwise, an inconsistency is reported to the Rust developer.

Note that the above approach to guarantee the correctness of the rectified programs is incomplete, because different execution

Table 3: Experiment Results on 36 Rust Projects with 5,108,432 Lines of Rust Code

Project	Description	Warnings	Bugs	Accuracy (%)
redox-os	an OS	10	6	60.0
tikv	a KV database	5	1	20.0
servo	a Web browser	2	1	50.0
actix-web	a web framework	2	0	0.0
deno	a game engine	2	2	100.0
citybound	a simulation game	2	2	100.0
nebulet	a microkernel	2	0	0.0
resvg	an SVG library	2	2	100.0
zcash	Zerocash protocol	2	0	0.0
Total	NA	29	14	48.3

trace may generate the same output. Thus, we also leveraged a trace validation approach, as we discussed in Section 3.6.

In our evaluation, all the rectifications to the 14 buffer overflow vulnerabilities identified by RUPAIR, are proved to be correct.

This experiment demonstrated the correctness of RUPAIR.

4.6 RQ4: Cost

To answer **RQ4**, for each rectified programs, we measure the size increment, the runtime overhead RUPAIR introduced into the programs being rectified, and the performance of RUPAIR.

Size Increment. For each rectified programs, RUPAIR inserted 15 extra MIR instructions, and 29.75 x86-64 assembly instructions, on average. Such size increments are insignificant.

Cost. To understand the runtime overhead RUPAIR introduced into the rectified programs, we further compare the execution time for each rectified program with its corresponding insecure program. In Table 4, we presented the execution time (in microsecond), along

Table 4: Execution Time for Programs before and after the Rectification (in millisecond)

CVE or project	Rectified?	Before	After	Loss (%)
CVE-2018-1000810	Yes	372	384	3.2
RUSTSEC-2017-0004	No	NA	NA	NA
redox-ralloc	Yes	3700	3846	3.9
redox-relibc	Yes	3813	3958	3.8
array-mul	Yes	10336	10551	2.1
array-sum	Yes	19589	19976	2.0
binary-search	Yes	8014	8202	2.3
find-duplicate	Yes	18352	18441	4.9
find-median	Yes	3702	3839	3.7
sorted-array-merge	Yes	5686	6070	6.7
Average	NA	NA	NA	3.6

with the performance loss. The loss L is calculated by

$$L = A/B - 1,$$

where A and B stand for execution time after and before the rectification, respectively. The loss is between 2.0% and 6.7%, with an average of 3.6% for all programs.

This experiment shows that the extra cost RUPAIR introduced is low and insignificant.

Performance. Furthermore, to gain an understanding of the performance of RUPAIR, we conducted experiments to measure the time RUPAIR spent in rectifying each of the Rust projects. In Figure 5, we presented the running time RUPAIR used to scan each of the

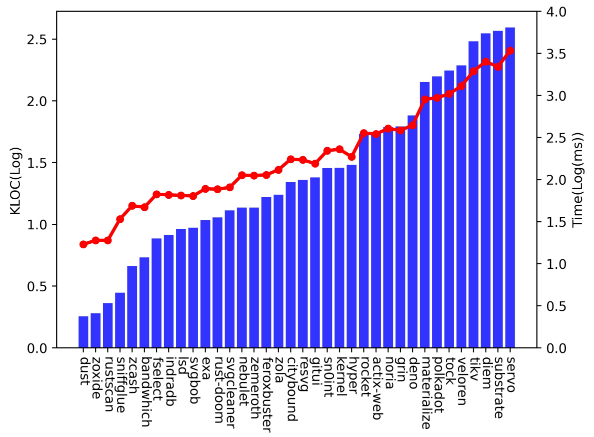

Figure 5: Performance of RUPAIR on the 36 Rust Projects

36 projects. The x−axis shows the projects, in an increasing order of code sizes. The y−axis on the left shows the sizes of each project (lines of code) in base-10 log scale. The y−axis on the right shows the execution time for each project (in milliseconds) also in base-10 log scale. This result demonstrated that the execution time of RUPAIR increases linearly to the sizes of programs. It takes RUPAIR 1853 milliseconds to analyze the 5,108,432 lines of Rust source code (2757 LOC per millisecond).

This experiment demonstrates that RUPAIR is efficient to analyze real-world large Rust projects.

5 DISCUSSION

In this section, we report some lessons learned during the design and implementation of RUPAIR, and present some future research directions. It should be noted that this work represents the first step towards detecting and rectifying buffer overflow bugs in real-world Rust projects, and demonstrates that it can effectively detect and rectify real CVEs and memory bugs.

5.1 Intra- and Inter-procedural Analysis

An intra-procedural program analysis operates on a procedure granularity, whereas an inter-procedural analysis analyze the whole program. We have 3 primary goals for RUPAIR: 1) it should be effective to identify and fix real-world CVEs; 2) it should be efficient to analyze large Rust projects; and 3) it should incur low cost. Experiment results have demonstrated the use of an intra-function analysis as in this work is successful in achieving these goals. On the other hand, it should be noted that the architecture of RUPAIR (Fig. 4) is neutral to any concrete analysis algorithm, as long as the algorithm works on Rust AST or MIR. To implement such an inter-procedural algorithm, one would build a call graph G, and visit graph edges reversely from the use-to-def in G. Although the use of an intra-procedural algorithm should not be considered as a limitation of this work, we believe it is worthy to explore the effectiveness of an inter-procedural algorithm in future work.

5.2 Organization of the `rustc` Compiler

The current `rustc` compiler leverages three primary intermediate representations: 1) AST (also called HIR), the high-level abstract syntax trees close to Rust program sources ; 2) MIR, a control-flow graph-based middle-level IR for program analysis and optimizations; and 3) LLVM, the low-level LLVM [45] byte code representation for target code generation. In an early design of RUPAIR, we have tried to perform the buffer overflow analysis and program rectification solely on the MIR representation, as it's straightforward to implement the analysis in this work by leveraging MIR's existing static analysis framework.

However, our implementation experience has demonstrated that such a design is not suitable to our purpose in this work. The key difficulty is that the MIR is relatively low-level, and does not carry the required source code information necessary for this study. To be specific, MIR does not maintain the `unsafe` code block information, which makes Algorithm 1 difficult to implement. Although the query feature of `rustc` make it possible to make use of 1) the AST, to obtain the necessary source program information; and 2) MIR, to implement the analysis algorithm, we believe the analysis will be easier to implement if MIR is annotated with necessary information in a future compiler version.

5.3 Other Rust Vulnerabilities

This paper only discussed the detection and rectification of buffer overflow memory bugs. Although memory bugs are pervasive and severe, existing studies have demonstrated Rust programs are also vulnerable to concurrency bugs [64]. Due to the distinct nature of memory and concurrency bugs, we leave the study of concurrency bugs to future work.

6 RELATED WORK

In recent years, the study of security and reliability of Rust language has drawn much research attentions, and there are a significant amount of research on automated program repair. However, the work in this paper stands for a novel contribution to these fields.

6.1 Rust Security

Security of Rust features. Evans et al. [31] performed a large-scale empirical study to explore how software developers are using `unsafe` Rust in real-world Rust libraries and applications. Xu et al.[76] performed an in-depth analysis regarding the culprits of real-world memory-safety bugs and extracted three typical categories. Qin et al.[64] conducted the first empirical study of safety practices and safety issues in real-world Rust programs and had a particular focus on how Rust ownership and lifetime rules impact developers. However, these studies only study some specific unsafe features, but don't consider automatically fixing bugs like our work.

Rust Semantics Formalization. Reed[66] presented a formal semantics for Rust that captures the key features relevant to memory safety. LAMQADEM et al.[43] presented a formalization of the Rust static semantics that includes lifetime inference. Wang et al.[72] designed a formal operational semantics of Rust capturing ownership, ownership moves and borrows and formalized the semantics in the K framework. CRUST [71] is a bounded model checker designed to verify the safety of Rust libraries implemented using unsafe code. RustBelt [39] provided the first formal safety proof for a realistic subset of Rust. Dang et al. [26] extended the RustBelt project and added support for the weak memory model widely used in the Rust library. Other researchers [34] [18] [30] conducted semi-automated verification on Rust programs using Viper [59], a verification platform based on separation logic. This approach has also been used to deal with unsafe code [18], generics and type traits [30]. Matsushita et al. [55] proposed a novel method for CHC-based program verification, and formalized the semantics for a core language of Rust. However, the focus of these studies in on formalizing the feature semantics, and do not discuss the identification and rectification of bugs.

Rust Security Tools. Zhang et al.[83] constructed a tool VRLifeTime to help programmers reason about lifetime-related errors. Luo et al. [65] built RustViz, a tool that generates an interactive timeline depicting ownership and borrowing events for each variable. Light [48] proposed Reenix, a Unix-like operating system kernel. Amit et al. [46] used Rust to develop a new embedded operating system for microcontrollers called Tock. Facebook posted Libra [12], which used Rust to implement its underlying blockchain. However, a major limitation of these studies is that none of these tools can identify or rectify buffer overflow bugs.

6.2 Automated Program Rectification

Automated program rectification has been the subject of recent study in the software engineering research community. And the research efforts can be divided into two categories: the search-based method and the semantic-based method.

The Search-based Program Rectification. GenProg [74] is one of the earliest work on search-based program repair technology, by using genetic programming to guide the generation and verification of patches. Weimer et al. [73] proposed an AE (Adaptive Equivalence) method to optimize GenProg, which identifies semantically equivalent patches based on approximate semantic equivalence relations, and reduces the number of candidate patches. Long et al. [52] proposed the SPR method, which generates patches through the parametric patch mode with the help of abstract values, which effectively reduces the number of candidate patches generated. Long et al. [53] further proposed the Prophet method to optimize the ranking of candidate patches and verify the correct patch first. Debroy and Wong [28] proposed a program repair method based on mutation testing, which generates patches with the help of mutation operators in mutation testing. Qi et al. [63] select a random search algorithm in the candidate patch search process to obtain a more effective patch search strategy. Kim et al. [41] proposed the strategy of mining open source projects along with the PAR method, and summarized code modification templates. Tan et al.

[70] suggested to use Anti patterns to prohibit conversion operations, so as not to modify the template to restrict the search space of candidate patches.

The Semantics-based Program Rectification. Nguyen et al. proposed SemFix [61] to use the Tarantula defect location method [37] to infer the sentences containing the defect and to fix them. Mechtaev et al. [56] proposed the DirectFix approach to improve the readability and comprehensibility of generated patches, by leveraging the program synthesis technique [35]. In order to repair larger-scale defective programs, Mechtaev et al. [57] proposed the Angelix, which uses lightweight constraints for code synthesis.

Domain-specific Program Rectification. Many studies apply automatic program repair methods to specific domains. For instance, CFix [36] and HFix [50] propose repair strategies for data races and order violations in concurrent programs. ConcBugAsssit [40] and DFixer [23] studied program defect repair methods for deadlock problems in concurrent programs. Cornu et al. [25] proposed NPEfix to fix null pointer exceptions in Java. Gao et al. [32] proposed LeakFix to detect and repair memory leaks in C programs.

However, all the above researches cannot be used directly for fixing Rust buffer overflow vulnerabilities, due to Rust's two safe and unsafe sub-languages, and its unique security features.

7 CONCLUSION

In this research, we propose Rupair, the first automated program rectification system to identify and fix buffer overflow bugs in Rust programs. The key novelty of Rupair is a data-flow based analysis algorithm which works across the safe and unsafe sub-languages of Rust. We conduct a number of experiments to apply Rupair on micro benchmarks of real CVEs and vulnerabilities, and on real-world Rust projects. The experiment results demonstrated that Rupair is effective in identifying and rectifying real buffer overflow bugs, including the previously unknown ones. In addition, Rupair is efficient and the cost introduced is low.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work is supported by a graduate education innovation program of USTC under grant No. 2020YCJC41, No. 2021YCJC34.

REFERENCES

[1] 2014. integer-overflow. https://rust-lang.github.io/rfcs/0560-integer-overflow.html.

[2] 2018. rust-moving. https://users.rust-lang.org/t/rust-mutability-moving-andborrowing-the-straight-dope/22166.

[3] 2020. nomicon-casts. https://doc.rust-lang.org/nomicon/casts.html.

[4] 2020. Parity. https://github.com/paritytech/parity-ethereum.

[5] 2020. The Rust CVEs. https://github.com/system-pclub/rust-study.

[6] 2020. rust-lifetime. https://doc.rust-lang.org/nomicon/lifetimes.html.

[7] 2020. rust-ownership. https://doc.rust-lang.org/nomicon/ownership.html.

[8] 2020. rust-survey-2020. https://blog.rustlang.org/2020/12/16/rust-survey-2020.html.

[9] 2020. TTstack. https://github.com/rustcc/TTstac.

[10] 2020. unsafe-rust. https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html.

[11] 2021. The AddressSanitizer. https://clang.llvm.org/docs/AddressSanitizer.html.

[12] 2021. Libra. https://www.diem.com/en-us/.

[13] 2021. Rust Specification. https://doc.rust-lang.org/.

[14] 2021. The Servo Browser Engine. https://servo.org/.

[15] 2021. TiKV. https://github.com/tikv/tikv.

[16] Hiralal Agrawal and Joseph R Horgan. 1990. Dynamic program slicing. *ACM SIGPlan Notices* 25, 6 (1990), 246–256.

[17] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley. https://www.worldcat.org/oclc/12285707

[18] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J Summers. 2019. Leveraging Rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

[19] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: learning to fix bugs automatically. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 159:1–159:27. https://doi.org/10.1145/3360585

[20] Clark Barrett and Cesare Tinelli. 2018. Satisfiability modulo theories. In *Handbook of Model Checking*. Springer, 305–343.

[21] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 171–177. https://doi.org/10.1007/978-3-642-22110-1_14

[22] David W Binkley and Keith Brian Gallagher. 1996. Program slicing. *Advances in computers* 43 (1996), 1–50.

[23] Yan Cai and Lingwei Cao. 2016. Fixing deadlocks via lock pre-acquisitions. In *Proceedings of the 38th international conference on software engineering.* 1109–1120.

[24] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9058)*, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 3–11. https://doi.org/10.1007/978-3-319-17524-9_1

[25] Benoit Cornu, Thomas Durieux, Lionel Seinturier, and Martin Monperrus. 2015. Npefix: Automatic runtime repair of null pointer exceptions in java. *arXiv preprint arXiv:1512.07423* (2015).

[26] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2019. RustBelt meets relaxed memory. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–29.

[27] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 337–340.

[28] Vidroha Debroy and W Eric Wong. 2010. Using mutation to automatically suggest fixes for faulty programs. In *2010 Third International Conference on Software Testing, Verification and Validation.* IEEE, 65–74.

[29] Yu Ding, Ran Duan, Long Li, Yueqiang Cheng, Yulong Zhang, Tanghui Chen, Tao Wei, and Huibo Wang. 2017. Poster: Rust SGX SDK: Towards memory safety in Intel SGX enclave. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* 2491–2493.

[30] Matthias Erdin, Vytautas Astrauskas, and Federico Poli. 2019. *Verification of Rust Generics, Typestates, and Traits.* Ph.D. Dissertation. Master¡¯s thesis, ETH Zürich.

[31] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is rust used safely by software developers?. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 246–257. https://doi.org/10.1145/3377811.3380413

[32] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. 2015. Safe memory-leak fixing for c programs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 459–470.

[33] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. 2018. K-Miner: Uncovering Memory Corruption in Linux. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018.* The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_05A-1_Gens_paper.pdf

[34] Florian Hahn. 2016. *Rust2Viper: Building a static verifier for Rust.* Master's thesis. ETH Zurich.

[35] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. IEEE, 215–224.

[36] Guoliang Jin, Wei Zhang, and Dongdong Deng. 2012. Automated concurrency-bug fixing. In *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12).* 221–236.

[37] James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering.* 273–282.

[38] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

[39] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–34.

[40] Sepideh Khoshnood, Markus Kusano, and Chao Wang. 2015. ConcBugAssist: constraint solving for diagnosis and repair of concurrency bugs. In *Proceedings of the 2015 international symposium on software testing and analysis*. 165–176.

[41] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 802–811.

[42] Shuvendu K Lahiri and Sanjit A Seshia. 2004. The UCLID decision procedure. In *International Conference on Computer Aided Verification*. Springer, 475–478.

[43] AMIN AIT LAMQADEM. 2019. A Formalization of the Static Semantics of Rust. (2019).

[44] Stefan Lankes, Jens Breitbart, and Simon Pickartz. 2019. Exploring rust for unikernel development. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*. 8–15.

[45] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE, 75–86.

[46] Amit Levy, Michael P Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. 2015. Ownership is theft: Experiences building an embedded OS in Rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*. 21–26.

[47] Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. The case for writing a kernel in rust. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*. 1–7.

[48] Alex Light. 2015. Reenix: Implementing a unix-like operating system in rust. *Undergraduate Honors Theses, Brown University* (2015).

[49] Per Lindgren, Nils Fitinghoff, and Jorge Aparicio. 2019. Cargo-call-stack Static Call-stack Analysis for Rust. In *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*, Vol. 1. IEEE, 1169–1176.

[50] Haopeng Liu, Yuxi Chen, and Shan Lu. 2016. Understanding and generating high quality patches for concurrency bugs. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 715–726.

[51] Peiming Liu, Gang Zhao, and Jeff Huang. 2020. Securing unsafe rust programs with XRust. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 234–245. https://doi.org/10.1145/3377811.3380325

[52] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 166–178.

[53] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 298–312.

[54] Gongming Luo, Vishnu Reddy, Marcelo Almeida, Yingying Zhu, Ke Du, and Cyrus Omar. 2020. RustViz: Interactively Visualizing Ownership and Borrowing. *CoRR* abs/2011.09012 (2020). arXiv:2011.09012 https://arxiv.org/abs/2011.09012

[55] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2020. RustHorn: CHC-based verification for Rust programs. In *European Symposium on Programming*. Springer, Cham, 484–514.

[56] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. Directfix: Looking for simple program repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 448–458.

[57] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*. 691–701.

[58] Samantha Miller, Kaiyuan Zhang, Mengqi Chen, Ryan Jennings, Ang Chen, Danyang Zhuo, and Thomas Anderson. 2021. High Velocity Kernel File Systems with Bento. In *19th {USENIX} Conference on File and Storage Technologies ({FAST} 21)*. 65–79.

[59] Peter Müller, Malte Schwerhoff, and Alexander J Summers. 2016. Viper: A verification infrastructure for permission-based reasoning. In *International conference on verification, model checking, and abstract interpretation*. Springer, 41–62.

[60] George C Necula. 2000. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. 83–94.

[61] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 772–781.

[62] Pengxiang Ning and Boqin Qin. 2020. Stuck-me-not: A deadlock detector on blockchain software in Rust. In *The 11th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN 2020) / The 10th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2020) / Affiliated Workshops, Madeira, Portugal, November 2-5, 2020 (Procedia Computer Science, Vol. 177)*, Elhadi M. Shakshuki and Ansar-Ul-Haque Yasar (Eds.). Elsevier, 599–604. https://doi.org/10.1016/j.procs.2020.10.085

[63] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*. 254–265.

[64] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding memory and thread safety practices and issues in real-world Rust programs. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 763–779. https://doi.org/10.1145/3385412.3386036

[65] Vishnu Reddy, Marcelo Almeida, Yingying Zhu, Ke Du, Cyrus Omar, et al. 2020. RustViz: Interactively Visualizing Ownership and Borrowing. *arXiv preprint arXiv:2011.09012* (2020).

[66] Eric Reed. 2015. 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.

[67] Martin C. Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee. 2004. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, Eric A. Brewer and Peter Chen (Eds.). USENIX Association, 303–316. http://www.usenix.org/events/osdi04/tech/rinard.html

[68] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. 2020. Intra-unikernel isolation with intel memory protection keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 143–156.

[69] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: brute force vulnerability discovery*. Pearson Education.

[70] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 727–738.

[71] John Toman, Stuart Pernsteiner, and Emina Torlak. 2015. Crust: a bounded verifier for rust (N). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 75–80.

[72] Feng Wang, Fu Song, Min Zhang, Xiaoran Zhu, and Jun Zhang. 2018. KRust: A Formal Executable Semantics of Rust. In *2018 International Symposium on Theoretical Aspects of Software Engineering, TASE 2018, Guangzhou, China, August 29-31, 2018*, Jun Pang, Chenyi Zhang, Jifeng He, and Jian Weng (Eds.). IEEE Computer Society, 44–51. https://doi.org/10.1109/TASE.2018.00014

[73] Westley Weimer, Zachary P Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 356–366.

[74] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 364–374.

[75] Mark Weiser. 1984. Program slicing. *IEEE Transactions on software engineering* 4 (1984), 352–357.

[76] Hui Xu, Zhuangbin Chen, Mingshen Sun, and Yangfan Zhou. 2020. Memory-Safety Challenge Considered Solved? An Empirical Study with All Rust CVEs. *CoRR* abs/2003.03296 (2020). arXiv:2003.03296 https://arxiv.org/abs/2003.03296

[77] Zeming Yu, Linhai Song, and Yiying Zhang. 2019. Fearless Concurrency? Understanding Concurrent Programming Safety in Real-World Rust Software. *CoRR* abs/1902.01906 (2019). arXiv:1902.01906 http://arxiv.org/abs/1902.01906

[78] Chao Zhang, Tielei Wang, Tao Wei, Yu Chen, and Wei Zou. 2010. IntPatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. In *European Symposium on Research in Computer Security*. Springer, 71–86.

[79] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. 2011. ConSeq: detecting concurrency bugs through sequential errors. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 251–264.

[80] Wei Zhang, Chong Sun, and Shan Lu. 2010. ConMem: detecting severe concurrency bugs through an effect-oriented approach. *ACM Sigplan Notices* 45, 3 (2010), 179–192.

[81] Yuyao Zhang, Siqi Ma, Juanru Li, Kailai Li, Surya Nepal, and Dawu Gu. 2020. Smartshield: Automatic smart contract protection made easy. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 23–34.

[82] Ziyi Zhang, Boqin Qin, Yilun Chen, Linhai Song, and Yiying Zhang. 2020. VRLifeTime - An IDE Tool to Avoid Concurrency and Memory Bugs in Rust. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 2085–2087. https://doi.org/10.1145/3372297.3420024

[83] Ziyi Zhang, Boqin Qin, Yilun Chen, Linhai Song, and Yiying Zhang. 2020. VRLifeTime–An IDE Tool to Avoid Concurrency and Memory Bugs in Rust. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2085–2087.